O'REILLY®

TURING 图灵程序设计丛书

# Flink基础教程

## Introduction to Apache Flink

### Stream Processing for Real Time and Beyond

Flink项目核心成员执笔

阿里巴巴资深技术专家悉心翻译




[美] 埃伦·弗里德曼 [希] 科斯塔斯·宙马斯 著

王绍翾 译



中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

# 译者介绍

## 王绍翾

阿里巴巴资深技术专家，Apache Flink Committer，淘宝花名“大沙”。毕业于北京大学信息科学技术学院，后取得加州大学圣地亚哥分校计算机工程博士学位。目前就职于阿里巴巴计算平台事业部，负责Flink SQL引擎及机器学习的相关开发。加入阿里巴巴之前，在Facebook开发分布式图存储系统TAO。曾多次拜访由Flink创始团队创办的公司data Artisans，并与其首席执行官科斯塔斯·宙马斯（本书作者之一）以及首席技术官斯蒂芬·尤恩有着广泛的合作。

# 数字版权声明

TURING 图灵程序设计丛书

# Flink基础教程

## Introduction to Apache Flink:
Stream Processing for Real Time and Beyond

[美] 埃伦·弗里德曼 [希] 科斯塔斯·宙马斯 著
王绍翾 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（ＣＩＰ）数据

Flink基础教程 / (美) 埃伦·弗里德曼
(Ellen Friedman) , (希) 科斯塔斯·宙马斯
(Kostas Tzoumas) 著 ; 王绍翾译. -- 北京 : 人民邮电
出版社, 2018.8
（图灵程序设计丛书）
ISBN 978-7-115-49006-3

Ⅰ. ①F… Ⅱ. ①埃… ②科… ③王… Ⅲ. ①数据处
理软件—教材 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2018)第172767号

## 内 容 提 要

近年来，流处理变得越来越流行。作为高度创新的开源流处理器，Flink 拥有诸多优势，包括容错性、高吞吐、低延迟，以及同时支持流处理和批处理的能力。本书分为 6 章，侧重于介绍 Flink 的核心设计理念、功能和用途，内容涉及事件时间和处理时间、窗口和水印机制、检查点机制、性能测评，以及 Flink 如何实现批处理。

本书面向有兴趣学习如何分析大规模流数据的读者。

◆ 著　　　[美] 埃伦·弗里德曼 [希] 科斯塔斯·宙马斯
译　　　王绍翾
责任编辑　谢婷婷
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：880×1230　1/32
印张：3
字数：114千字　　　　2018年 8 月第 1 版
印数：1 – 3 000册　　　　2018年 8 月北京第 1 次印刷
著作权合同登记号　图字：01-2018-4238号

定价：39.00元
**读者服务热线：(010)51095186转600　印装质量热线：(010)81055316**
**反盗版热线：(010)81055315**
**广告经营许可证：京东工商广登字 20170147 号**

# 版权声明

# O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，O’Reilly 的每一项产品都反映了公司不可动摇的理念——信息是激发创新的力量。

## 业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

# 目录

# 前言

近几年，许多人开始对如何分析大规模系统中的流数据感兴趣，部分原因是，在某些场景下对实时数据进行实时分析显得非常有价值和吸引力。然而，通过低延迟的应用程序及时获得有用的信息，只是高性能流处理带来的众多好处之一。

本书介绍的 Apache Flink（以下简称 Flink）作为一种高度创新的开源流处理器，具备惊人的潜力，能够帮助你在以流为基础的各种计算中获益。Flink 不仅可以真正实现实时的容错性分析，还可以分析历史数据，并且极大地简化数据处理流程。最让人惊喜的是，Flink 用同一种底层技术来实现流处理和批处理。它拥有完备的语义和强大的性能，这使得应用程序的开发变得简单，其架构也使得应用程序的维护变得容易。

本书将全面介绍 Flink 的功能，并且讲解常见的使用方法，包括如何在生产环境中使用它。Flink 社区由来自世界各地的开发人员和用户组成，整个社区十分活跃，并且成长迅速。第一届 Flink 专属研讨会定名为 Flink Forward，于 2015 年 10 月在德国柏林举行，第二届于 2016 年 9 月举行。还有各种线下聚会在全球范围内举行，新的 Flink 用例在聚会中被大家广泛讨论。

## 如何阅读本书

本书对技术人员和非技术人员都有帮助。对于本书所讲解的设计理念和功能，你并不需要具备特殊技能或者拥有流处理经验就能理解，但是如果对

大数据系统有一定的了解，将会使阅读获得更好的效果。如果需要尝试运行本书中的示例代码，则需要具备 Java 或者 Scala 的经验。本书会清楚地讲解示例背后的核心概念，即使不懂代码也并不影响阅读。

第 1~3 章阐述 Flink 是基于哪些需求被开发出来的，以及它如何满足这些需求；还会介绍流处理架构的优势，以及 Flink 的整体设计。第 4 章至附录对 Flink 的功能进行更深层的技术性阐释。

# 排版约定



该图标表示一般性注解。



该图标表示提示或建议。

# 电子书

扫描如下二维码，即可购买本书电子版。

第 1 章

# 为何选择Flink

> 人们对某件事的正确理解往往来自基于有效论据的结论。要获得这样的结论，最有效的方法就是沿着事件发生的轨迹进行分析。

许多系统都会产生连续的事件流，如行驶中的汽车发射出 GPS 信号，金融交易，移动通信基站与繁忙的智能手机进行信号交换，网络流量，机器日志，工业传感器和可穿戴设备的测量结果，等等。如果能够高效地分析大规模流数据，我们对上述系统的理解将会更清楚、更快速。简而言之，流数据更真实地反映了我们的生活方式。

因此，我们自然希望将数据用事件流的方式收集起来并加以处理。但直到目前，这并不是整个行业的标准做法。流处理并非全新的概念，但它确实是一项专业性强且极具挑战性的技术。实际上，企业常见的数据架构仍旧假设数据是有头有尾的有限集。这个假设存在的大部分原因在于，与有限集匹配的数据存储及处理系统建起来比较简单。但是，这样做无疑给那些天然的流式场景人为地加了限制。

我们渴望按照流的方式处理数据，但要做好很困难；随着大规模数据在各行各业中出现，难度越来越大。这是一个属于物理学范畴的难题：在大型分布式系统中，数据一致性和对事件发生顺序的理解必然都是有限的。伴随着方法和技术的演化，我们尽可能使这种局限性不危及商业目标和运营目标。

在这样的背景下，Apache Flink（以下简称 Flink）应运而生。作为在公共社区中诞生的开源软件，Flink 为大容量数据提供流处理，并用同一种技术实现批处理。

在 Flink 的开发过程中，开发人员着眼于避免其他流处理方法不得不在高效性或者易用性方面所做的妥协。

本书将讨论流处理的一些潜在好处，从而帮助你确定以流为基础的数据处理方法是否适合你自己的商业目标。流处理的一些数据来源以及适用场景可能会让你感到意外。此外，本书还将帮助你理解 Flink 的技术以及这些技术如何克服流处理面临的困难。

本章将介绍人们希望通过分析流数据获得什么，以及在大规模流数据分析过程中面临的困难。本章是关于 Flink 的入门介绍，你可以看到人们平常（包括在生产环境中）是怎么使用它的。

## 1.1 流处理欠佳的后果

谁需要和流数据打交道呢？首先映入脑海的是从事传感器测量和金融交易的工作人员。对于他们来说，流处理非常有用。但是流数据来源非常广泛，两个常见的例子是：网站获得的能够反映用户行为的点击流数据，以及私有数据中心的机器日志。事实上，流数据来源无处不在，但是从连续事件中获得数据并不意味着可以在批量计算中使用这些数据。如今，处理大规模流数据的新技术正在改变这一状况。

如果说处理大规模流数据是一个历史性难题，我们为什么还要不厌其烦地尝试打造更好的流处理系统呢？在介绍支持流处理的新架构及新技术之前，我们先来谈谈不能很好地处理流数据会有什么后果。

### 1.1.1 零售业和市场营销

在现代零售业中，网站点击量就代表了销量。网站获得的点击数据可能是大量、连续、不均匀的。用以往的技术很难处理好如此规模的数据。仅是构建批量系统处理这些数据流[1]就很有挑战性：结果很可能是需要一个庞大

注 1：在本书中，“数据流”是指由连续数据组成的流；“流数据”是指数据流中的数据。
——译者注

且复杂的系统。并且，传统的做法还会带来数据丢失、延迟、错误的聚合结果等问题。这样的结果怎能对商业领域有所帮助呢？

假设你正在向首席执行官汇报上一季度的销售数据，你肯定不想事后因为使用了不准确的数据而不得不向首席执行官更正汇报结果。如果不能良好地处理点击数据，你很可能对网站点击量进行不准确的计算，这将导致广告投放报价和业绩数字不准确。

航空旅客服务业面临同样的挑战：航空公司需要快速、准确地处理从各种渠道获得的大量数据。例如，当为一名旅客办理登机手续时，需要对该旅客的机票预订数据进行核对，还需要核对行李处理信息、航班状态信息和账单信息。如果没有强大的技术来支持流处理，这种规模的数据是很难不出错的。近几年，美国四大航空公司中有三家都出现了大面积的服务中断，这几次故障都可以归咎于大规模实时数据处理失败。

当然，很多相关问题（如怎样避免重复预订酒店或演唱会门票），一般都能够通过有效的数据库操作来解决，但是这种操作相当费钱，也费精力。尤其当数据量增加时，成本会飙升，并且在某些情况下，数据库的反应速度会变得特别慢。由于缺乏灵活性，开发速度受到影响，项目在庞大又复杂或者不断发生变化的系统中进展缓慢。想要在大型系统中处理流数据，并且在保持一致性的同时有效地控制成本，难度非常大。

幸运的是，现代的流处理器经常可以用新的方式解决这些问题，这使得实时处理大规模数据的成本更低。流处理还激发了新的尝试，比如构建一个系统，该系统能够基于顾客当下购买的商品实时给出相关的建议，看看他们是否还需要买一些别的商品。这不代表流处理器替代了数据库（远远不能替代），而是说在数据库处理不好时，流处理器提供了更好的解决方案。这样做也使数据库得以解脱，不用再参与对当前业务状态的实时分析。第 2 章在介绍流处理架构时将对这一转变做更深入的讲解。

### 1.1.2　物联网

物联网是流数据被普遍应用的领域。在物联网中，低延迟的数据传输和处理，以及准确的数据分析通常很关键。各类仪器中的传感器频繁地获得测量数据，并将它们以流的形式传输至数据中心。在数据中心内，实时或者

接近实时的应用程序将更新显示板，运行机器学习模型，发布警告，并就许多不同的服务项目提供反馈。

交通运输业也体现了流处理的重要性。举例来说，先进的列车系统依靠的是传感器测量数据，这些数据从轨道传至列车，再从列车传至沿途的传感器；与此同时，报告也被发送回控制中心。测量数据包括列车的速度和位置，以及轨道周边的状况。如果流数据没有被正确处理，调整意见和警告就不能相应产生，从而也就不能通过对危险状况做出反应来避免事故发生。

另一个例子是“智能”汽车，或称联网汽车，它们通过移动网络将数据传输回制造商。在有些国家（北欧国家、法国和英国，美国则刚开始），联网汽车甚至可以将信息传给保险公司；如果是赛车，信息还可以通过射频链路传送至维修站进行分析。此外，一些智能手机应用程序还支持数百万司机共享实时路况信息。



图 1-1：许多情况都需要考虑数据的时效性，包括使用物联网数据的交通运输业。供数百万司机共享的实时路况信息依靠的是对流数据及时地进行合理和准确的分析（图片来源：©2016 弗里德曼）

物联网对公用事业也有影响。相关公司已经开始安装智能计量表，以替换每个月需要人工读数的旧表。智能计量表可以定期将用电量反馈给公司（例如每 15 分钟一次）。有些公司正在尝试每 30 秒就进行一次测量。使用

智能计量表的这一转变带来了大量的流数据，同时也获得了大量的潜在收益。其中一个好处就是通过机器学习模型来检测设备故障或者窃电等使用异常。如果不能对流数据进行高吞吐、低延迟和准确的处理，这些新的目标都无法实现。

如果流处理做得不好，其他物联网项目也会遭殃。大型设备，比如风力涡轮机、生产设备和钻井泵，都依赖对传感器测量数据的分析来获得故障警告。如果不能及时地处理好这些设备的流数据，将可能付出高昂的代价，甚至导致灾难性后果。

## 1.1.3 电信业

电信业是一个特殊的例子，它广泛地应用了基于各种目的而产生的跨地域的事件流数据。如果电信公司不能很好地处理流数据，就不能在某个移动通信基站出现流量高峰前预先将流量分配给其他的基站，也不能在断电时快速做出反应。通过处理流数据来进行异常检测，如检测通话中断或者设备故障，对于电信业来说至关重要。

## 1.1.4 银行和金融业

因为流处理做得不好而给银行以及金融业带来的潜在问题是极其显著的。从事零售业务的银行不希望客户交易被延迟或者因为错误统计而造成账户余额出错。曾有一个说法叫作“银行家工作时间”，指的就是银行需要在下午早早关门进行结算，这样才能保证第二天营业之前算出准确的账。这种批量作业的营业模式早已消失。如今，交易和报表都必须快速且准确地生成；有些新兴的银行甚至提供实时的推送通知，以及随时随地访问手机银行的服务。在全球化经济中，能够提供 24 小时服务变得越来越重要。

那么，如果缺少能够灵敏地实时检测出用户行为异常的应用程序，会对金融机构带来什么后果呢？信用卡欺诈检测需要及时的监控和反馈。对异常登录的检测能发现钓鱼式攻击，从而避免巨大的损失。



在许多情况下，人们希望用低延迟或者实时的流处理来获得数据的高时效性，前提是流处理本身是准确且高效的。

## 1.2 连续事件处理的目标

能够以非常低的延迟处理数据，这并不是流处理的唯一优势。人们希望流处理不仅做到低延迟和高吞吐，还可以处理中断。优秀的流处理技术应该能使系统在崩溃之后重新启动，并且产出准确的结果；换句话说，优秀的流处理技术可以容错，而且能保证 exactly-once[2]。

与此同时，获得这种程度的容错性所采用的技术还需要在没有数据错误的情况下不产生太大的开销。这种技术需要能够基于事件发生的时间（而不是随意地设置处理间隔）来保证按照正确的顺序跟踪事件。对于开发人员而言，不论是写代码还是修正错误，系统都要容易操作和维护。同样重要的是，系统生成的结果需要与事件实际发生的顺序一致，比如能够处理乱序事件流（一个很不幸但无法避免的事实），以及能够准确地替换流数据（在审计或者调试时很有用）。

## 1.3 流处理技术的演变

分开处理连续的实时数据和有限批次的数据，可以使系统构建工作变得更加简单，但是这种做法将管理两套系统的复杂性留给了系统用户：应用程序的开发团队和 DevOps 团队需要自己使用并管理这两套系统。

为了处理这种情况，有些用户开发出了自己的流处理系统。在开源世界里，Apache Storm 项目（以下简称 Storm）是流处理先锋。Storm 最早由 Nathan Marz 和创业公司 BackType（后来被 Twitter 收购）的一个团队开发，后来才被 Apache 软件基金会接纳。Storm 提供了低延迟的流处理，但是它为实时性付出了一些代价：很难实现高吞吐，并且其正确性没能达到通常所需的水平。换句话说，它并不能保证 exactly-once；即便是它能够保证的正确性级别，其开销也相当大。

注 2：对 exactly-once 的解释，详见 5.1 节。——编者注

### Lambda 架构概述：优势和局限性

对低成本规模化的需求促使人们开始使用分布式文件系统，例如 HDFS 和基于批量数据的计算系统（MapReduce 作业）。但是这种系统很难做到低延迟。用 Storm 开发的实时流处理技术可以帮助解决延迟性的问题，但并不完美。其中的一个原因是，Storm 不支持 exactly-once 语义，因此不能保证状态数据的正确性，另外它也不支持基于事件时间的处理。有以上需求的用户不得不在自己的应用程序代码中加入这些功能。

后来出现了一种混合分析的方法，它将上述两个方案结合起来，既保证低延迟，又保障正确性。这个方法被称作 Lambda 架构，它通过批量 MapReduce 作业提供了虽有些延迟但是结果准确的计算，同时通过 Storm 将最新数据的计算结果初步展示出来。

Lambda 架构是构建大数据应用程序的一种很有效的框架，但它还不够好。举例来说，基于 MapReduce 和 HDFS 的 Lambda 系统有一个长达数小时的时间窗口，在这个窗口内，由于实时任务失败而产生的不准确的结果会一直存在。Lambda 架构需要在两个不同的 API（application programming interface，应用程序编程接口）中对同样的业务逻辑进行两次编程：一次为批量计算的系统，一次为流式计算的系统。针对同一个业务问题产生了两个代码库，各有不同的漏洞。这种系统实际上非常难维护。

若要依靠多个流事件来计算结果，必须将数据从一个事件保留到下一个事件。这些保存下来的数据叫作计算的状态。准确处理状态对于计算结果的一致性至关重要。在故障或中断之后能够继续准确地更新状态是容错的关键。

在低延迟和高吞吐的流处理系统中维持良好的容错性是非常困难的，但是为了得到有保障的准确状态，人们想出了一种替代方法：将连续事件中的流数据分割成一系列微小的批量作业。如果分割得足够小（即所谓的微批处理作业），计算就几乎可以实现真正的流处理。因为存在延迟，所以不可能做到完全实时，但是每个简单的应用程序都可以实现仅有几秒甚至几亚秒的延迟。这就是在 Spark 批处理引擎上运行的 Apache Spark Streaming

（以下简称 Spark Streaming）所使用的方法。

更重要的是，使用微批处理方法，可以实现 exactly-once 语义，从而保障状态的一致性。如果一个微批处理作业失败了，它可以重新运行。这比连续的流处理方法更容易。Storm Trident 是对 Storm 的延伸，它的底层流处理引擎就是基于微批处理方法来进行计算的，从而实现了 exactly-once 语义，但是在延迟性方面付出了很大的代价。

然而，通过间歇性的批处理作业来模拟流处理，会导致开发和运维相互交错。完成间歇性的批处理作业所需的时间和数据到达的时间紧密耦合，任何延迟都可能导致不一致（或者说错误）的结果。这种技术的潜在问题是，时间由系统中生成小批量作业的那一部分全权控制。Spark Streaming 等一些流处理框架在一定程度上弱化了这一弊端，但还是不能完全避免。另外，使用这种方法的计算有着糟糕的用户体验，尤其是那些对延迟比较敏感的作业，而且人们需要在写业务代码时花费大量精力来提升性能。

为了实现理想的功能，人们继续改进已有的处理器（比如 Storm Trident 的开发初衷就是试图克服 Storm 的局限性）。当已有的处理器不能满足需求时，产生的各种后果则必须由应用程序开发人员面对和解决。以微批处理方法为例，人们往往期望根据实际情况分割事件数据，而处理器只能根据批量作业时间（恢复间隔）的倍数进行分割。当灵活性和表现力都缺乏的时候，开发速度变慢，运维成本变高。

于是，Flink 出现了。这一数据处理器可以避免上述弊端，并且拥有所需的诸多功能，还能按照连续事件高效地处理数据。Flink 的一些功能如图 1-2 所示。

与 Storm 和 Spark Streaming 类似，其他流处理技术同样可以提供一些有用的功能，但是没有一个像 Flink 那样功能如此齐全。举例来说，Apache Samza（以下简称 Samza）是早期的一个开源流处理器，它不仅没能实现 exactly-once 语义，而且只能提供底层的 API；同样，Apache Apex 提供了与 Flink 相同的一些功能，但不全面（比如只提供底层的 API，不支持事件时间，也不支持批量计算）。这些项目没有一个能和 Flink 在开源社区的规模上相提并论。

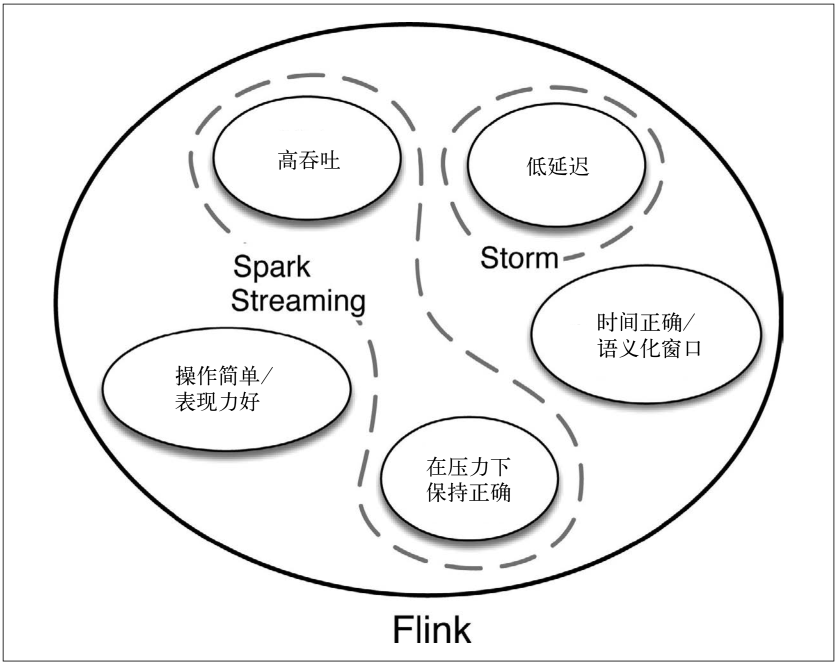



图 1-2：Flink 的一个优势是，它拥有诸多重要的流式计算功能。其他项目为了实现这些功能，都不得不付出代价。比如，Storm 实现了低延迟，但是在作者撰写本书时还做不到高吞吐，也不能在故障发生时准确地处理计算状态；Spark Streaming 通过采用微批处理方法实现了高吞吐和容错性，但是牺牲了低延迟和实时处理能力，也不能使窗口与自然时间相匹配，并且表现力欠佳

下面来了解 Flink 是什么，以及它是如何诞生的。

## 1.4 初探Flink

Flink 的主页[3] 在其顶部展示了该项目的理念："Apache Flink 是为分布式、高性能、随时可用以及准确的流处理应用程序打造的开源流处理框架。" Flink 不仅能提供同时支持高吞吐和 exactly-once 语义的实时计算，还能提供批量数据处理，这让许多人感到吃惊。鱼与熊掌并非不可兼得，Flink 用同一种技术实现了两种功能。

注 3：http://flink.apache.org

这个顶级的 Apache 项目是怎么诞生的呢？ Flink 起源于 Stratosphere 项目，Stratosphere 是在 2010~2014 年由 3 所地处柏林的大学和欧洲的一些其他的大学共同进行的研究项目。当时，这个项目已经吸引了一个较大的社区，一部分原因是它出现在了若干公共开发者研讨会上，比如在柏林举办的 Berlin Buzzwords，以及在科隆举办的 NoSQL Matters，等等。强大的社区基础是这个项目适合在 Apache 软件基金会中孵化的一个原因。

2014 年 4 月，Stratosphere 的代码被复制并捐献给了 Apache 软件基金会，参与这个孵化项目的初始成员均是 Stratosphere 系统的核心开发人员。不久之后，创始团队中的许多成员离开大学并创办了一个公司来实现 Flink 的商业化，他们为这个公司取名为 data Artisans。在孵化期间，为了避免与另一个不相关的项目重名，项目的名称也发生了改变。Flink 这个名字被挑选出来，以彰显这种流处理器的独特性：在德语中，flink 一词表示快速和灵巧。项目采用一只松鼠的彩色图案作为 logo，这不仅因为松鼠具有快速和灵巧的特点，还因为柏林的松鼠有一种迷人的红棕色。



图 1-3：左侧：柏林的红松鼠拥有可爱的耳朵；右侧：Flink 的松鼠 logo 拥有可爱的尾巴，尾巴的颜色与 Apache 软件基金会的 logo 颜色相呼应。这是一只 Apache 风格的松鼠！

这个项目很快完成了孵化，并在 2014 年 12 月一跃成为 Apache 软件基金会的顶级项目。作为 Apache 软件基金会的 5 个最大的大数据项目之一，Flink 在全球范围内拥有 200 多位开发人员，以及若干公司中的诸多上线场景，有些甚至是世界 500 强的公司。在作者撰写本书的时候，共有 34 个 Flink

线下聚会在世界各地举办，社区大约有 12 000 名成员，还有众多 Flink 演讲者参与到各种大数据研讨会中。2015 年 10 月，第一届 Flink Forward 研讨会在柏林举行。

## 批处理与流处理

Flink 是如何同时实现批处理与流处理的呢？答案是，Flink 将批处理（即处理有限的静态数据）视作一种特殊的流处理。

Flink 的核心计算构造是图 1-4 中的 Flink Runtime 执行引擎，它是一个分布式系统，能够接受数据流程序并在一台或多台机器上以容错方式执行。Flink Runtime 执行引擎可以作为 YARN（Yet Another Resource Negotiator）的应用程序在集群上运行，也可以在 Mesos 集群上运行，还可以在单机上运行（这对于调试 Flink 应用程序来说非常有用）。

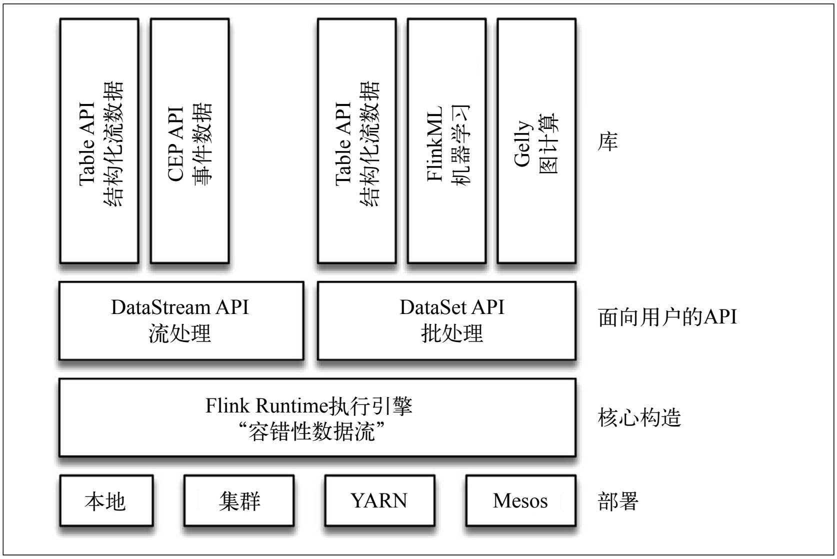



图 1-4：Flink 技术栈的核心组成部分。值得一提的是，Flink 分别提供了面向流处理的接口（DataStream API）和面向批处理的接口（DataSet API）。因此，Flink 既可以完成流处理，也可以完成批处理。Flink 支持的拓展库涉及机器学习（FlinkML）、复杂事件处理（CEP），以及图计算（Gelly），还有分别针对流处理和批处理的 Table API

能被 Flink Runtime 执行引擎接受的程序很强大，但是这样的程序有着冗长的代码，编写起来也很费力。基于这个原因，Flink 提供了封装在 Runtime 执行引擎之上的 API，以帮助用户更方便地生成流式计算程序。Flink 提供了用于流处理的 DataStream API 和用于批处理的 DataSet API。值得注意的是，尽管 Flink Runtime 执行引擎是基于流处理的，但是 DataSet API 先于 DataStream API 被开发出来，这是因为工业界对无限流处理的需求在 Flink 诞生之初并不大。

DataStream API 可以流畅地分析无限数据流，并且可以用 Java 或者 Scala 来实现。开发人员需要基于一个叫 DataStream 的数据结构来开发，这个数据结构用于表示永不停止的分布式数据流。

Flink 的分布式特点体现在它能够在成百上千台机器上运行，它将大型的计算任务分成许多小的部分，每个机器执行一个部分。Flink 能够自动地确保在发生机器故障或者其他错误时计算能持续进行，或者在修复 bug 或进行版本升级后有计划地再执行一次。这种能力使得开发人员不需要担心失败。Flink 本质上使用容错性数据流，这使得开发人员可以分析持续生成且永远不结束的数据（即流处理）。

Flink 解决了许多问题，比如保证了 exactly-once 语义和基于事件时间的数据窗口。开发人员不再需要在应用层解决相关问题，这大大地降低了出现 bug 的概率。

因为不用再在编写应用程序代码时考虑如何解决问题，所以工程师的时间得以充分利用，整个团队也因此受益。好处并不局限于缩短开发时间，随着灵活性的增加，团队整体的开发质量得到了提高，运维工作也变得更容易、更高效。Flink 让应用程序在生产环境中获得良好的性能。尽管相对较新，但是 Flink 已经在生产环境中得到了应用，下一节将做更详细的介绍。

## 1.5　生产环境中的Flink

本章旨在探讨为何选择 Flink。一个好的方法是听听在生产环境中使用 Flink 的开发人员解释他们选择 Flink 的原因，以及如何使用它。

### 1.5.1 布衣格电信

布衣格电信（Bouygues Telecom）是法国第三大移动通信运营商，隶属世界 500 强企业布衣格集团。布衣格电信使用 Flink 来进行实时事件处理，每天不间断地分析数十亿条消息。2015 年 6 月，在为 data Artisans 的博客撰写的文章中，Mohamed Amine Abdessemed[4] 描述了布衣格电信的目标以及 Flink 为什么能实现这些目标。

> ……布衣格电信最终选择了 Flink，因为它支持真正的流处理——通过上层的 API 和下层的执行引擎都能实时进行流处理，这满足了我们对可编程性和低延迟的需求。此外，使用 Flink，我们的系统得以快速上线，这是其他任何一种方案都做不到的。如此一来，我们就有了更多的人手开发新的业务逻辑。

Mohamed Amine Abdessemed 在于 2015 年 10 月举行的 Flink Forward 研讨会上也做了报告。布衣格电信试图给其工程师实时提供关于用户体验的反馈，让他们了解公司遍布全球的网络正在发生什么，并从网络的演进和运行的角度掌握发展动向。

为了实现这个目标，他们的团队搭建了一个用来分析网络设备日志的系统，它定义了衡量用户体验质量的各项指标。该系统每天处理 20 亿次事件，要求端到端延迟不超过 200 毫秒（包括由传输层负责的消息发布和由 Flink 操作的数据处理）。这些都在一个仅有 10 个节点（每个节点 1GB 内存）的小集群上完成。布衣格电信还希望这些经过部分处理的数据能被复用，从而在互不干扰的前提下满足各种商业智能分析需求。

该公司打算利用 Flink 的流处理能力来转换和挖掘数据。加工后的数据被推送回消息传输系统，以保证数据可以被不同的用户使用。

相比于其他处理方案，比如在数据进入消息队列之前进行处理，或者将数据分派给消费同一个消息队列的多个应用程序来分头处理，Flink 的处理方案更合适。

布衣格电信利用 Flink 的流处理能力完成了数据处理和数据迁移，它既满足了低延迟的要求，又具有很高的可靠性、可用性，以及易用性。Flink 为调

注 4：他在布衣格电信负责技术系统。——编者注

试工作提供了极大的便利，甚至支持切换到本地进行调试。它也支持程序可视化，有利于理解程序的运行原理。此外，Flink 的 API 吸引了很多开发人员和数据科学家。Mohamed Amine Abdessemed 在其文章中还提及布衣格电信的其他团队使用 Flink 解决了不同的问题。

### 1.5.2 其他案例

#### King公司

King 公司的游戏非常流行，全世界几乎每时每刻都有人在玩它的在线游戏。作为在线娱乐行业的佼佼者，该公司称自己已经开发了 200 多款游戏，市场覆盖 200 多个国家和地区。

King 公司的工程师曾在一篇博客文章中写道："我们每月有超过 3 亿的独立用户，每天从不同的游戏和系统中收到 300 亿次事件，基于这么大的数据量做任何流分析都是真正的技术挑战。因此，为我们的数据分析师开发工具来处理如此大规模的流数据，同时保证数据在应用中具有最大的灵活性，这些对于公司而言至关重要。"

King 公司用 Flink 构建的系统让其数据分析师得以实时地获取大量的流数据。Flink 的成熟度给他们留下了深刻的印象。即使面对像 King 公司这样复杂的应用环境，Flink 也能很好地提供支持。

#### Zalando公司

作为欧洲领先的在线时尚平台，Zalando 公司在全球拥有超过 1600 万的客户。该公司的网站将其组织结构描述为"多个敏捷、自主的小型团队"（换句话说，该公司采用了微服务架构）。

流处理架构为微服务提供了良好的支持。因此，Flink 提供的流处理能力满足了这种工作模式的需求，特别是支持业务流程监控和持续的 ETL[5] 过程。

#### Otto集团

Otto 集团是全球第二大 B2C（business to consumer，企业对顾客电子商务）在线零售商，也是欧洲时尚和生活领域最大的 B2C 在线零售商。

它的商业智能部门在最初开始评估开源流处理平台时，没有找到一种能够

注 5：ETL 是 Extract、Transform 和 Load 的缩写，即抽取、转换和加载。——编者注

符合其要求的平台，所以后来决定开发自己的流处理引擎。但是当试过 Flink 之后，该部门发现 Flink 满足了他们对流处理的所有需求，包括对众包用户代理的鉴定，以及对检索事件的辨识。

ResearchGate
从活跃用户的数量上看，ResearchGate 是最大的学术社交网络。它从 2014 年开始使用 Flink 作为其数据基础设施的一个主要工具，负责批处理和流处理。

**阿里巴巴集团**
阿里巴巴这个庞大的电子商务集团为买方和卖方提供平台。其在线推荐功能是通过基于 Flink 的系统 Blink 实现的。用户当天所购买的商品可以被用作在线推荐的依据，这是使用像 Flink 这样真正意义上的流处理引擎能够带来的好处之一。并且，这在那些用户活跃度异常高的特殊日期（节假日）尤其重要，也是高效的流处理相较于批处理的优势之一。

## 1.6 Flink的适用场景

本章开头提出了“为何选择 Flink”这一问题。比这个问题更大的则是“为何要用流数据？”本章解释了一些原因，比如在许多情况下，我们都需要观察和分析连续事件产生的数据。与其说流数据是特别的，倒不如说它是自然的——只不过从前我们没有流处理能力，只能做一些特殊的处理才能真正地使用流数据，比如将流数据攒成批量数据再处理，不然无法进行大规模的计算。使用流数据并不新鲜，新鲜的是我们有了新技术，从而可以大规模、灵活、自然和低成本地使用它们。

Flink 并不是唯一的流处理工具。人们正在开发和改进多种新兴的技术，以满足流处理需求。显然，任何一个团队选择某一种技术都是出于多方面的考虑，包括团队成员的已有技能。但是 Flink 的若干优点、易用性，以及使用它所带来的各种好处，使它变得非常有吸引力。另外，不断壮大且非常活跃的 Flink 社区也暗示着它值得一试。你会发现“为何选择 Flink”这个问题变成了“为何不选择 Flink 呢？”

在深入探讨 Flink 的工作原理之前，我们先来通过第 2 章了解如何设计数据架构才能从流处理中充分获益，以及流处理架构是如何带来诸多好处的。

# 第 2 章

# 流处理架构

数据架构设计领域正在发生一场变革，其影响不仅限于实时或近实时的项目。这场变革将基于流的数据处理流程视为整个架构设计的核心，而不是只作为某些专业化工作的基础。了解为何向流处理架构转变，可以帮助我们理解 Flink 和它在现代数据处理中所扮演的角色。

作为新型系统，Flink 扩展了“流处理”这个概念的范围。有了它，流处理不仅指实时、低延迟的数据分析，还指各类数据应用程序。其中，有些应用程序基于流处理器实现，有些基于批处理器实现，有些甚至基于事务型数据库实现。

事实证明，让 Flink 能有效工作的数据架构，恰恰是充分利用流数据的基础。为了帮助你理解，本书将详细介绍如何构建支持 Flink 流处理的管道。在这之前，我们先来看看与传统架构相比，流处理架构有何优势。

## 2.1 传统架构与流处理架构

对于后端数据而言，典型的传统架构是采用一个中心化的数据库系统，该系统用于存储事务性数据。换句话说，数据库（SQL 或者 NoSQL）拥有“新鲜”（或者说“准确”）的数据，这些数据反映了当前的业务状态，如系统当前有多少已登录的用户，网站当前有多少活跃用户，以及当前每个用

户的账户余额是多少。需要新鲜数据的应用程序都依靠数据库实现。分布式文件系统则用来存储不需要经常更新的数据，它们也往往是大规模批量计算所依赖的数据存储方式。

这种传统架构成功地服务了几十年，但随着大型分布式系统中的计算复杂度不断上升，这种架构已经不堪重负。许多公司经常遇到以下问题。

- 在许多项目中，从数据到达到数据分析所需的工作流程太复杂、太缓慢。
- 传统的数据架构太单一：数据库是唯一正确的数据源，每一个应用程序都需要通过访问数据库来获得所需的数据。
- 采用这种架构的系统拥有非常复杂的异常问题处理方法。当出现异常问题时，很难保证系统还能很好地运行。

传统架构的另一个问题是，需要通过在大型分布式系统中不断地更新来维持一致的全局状态。随着系统规模扩大，维持实际数据与状态数据间的一致性变得越来越困难；流处理架构则少了对这方面的要求，只需要维持本地的数据一致性即可。

作为一种新的选择，流处理架构解决了企业在大规模系统中遇到的诸多问题。以流为基础的架构设计让数据记录持续地从数据源流向应用程序，并在各个应用程序间持续流动。没有一个数据库来集中存储全局状态数据，取而代之的是共享且永不停止的流数据，它是唯一正确的数据源，记录了业务数据的历史。在流处理架构中，每个应用程序都有自己的数据，这些数据采用本地数据库或分布式文件进行存储。

## 2.2 消息传输层和流处理层

如何有效地实现流处理架构并从 Flink 中获益呢？一个常见的做法是设置消息传输层和流处理层，如图 2-1 所示。

(1) **消息传输层**从各种数据源（生产者）采集连续事件产生的数据，并传输给订阅了这些数据的应用程序和服务（消费者）。

(2) **流处理层**有 3 个用途：①持续地将数据在应用程序和系统间移动；②聚合并处理事件；③在本地维持应用程序的状态。

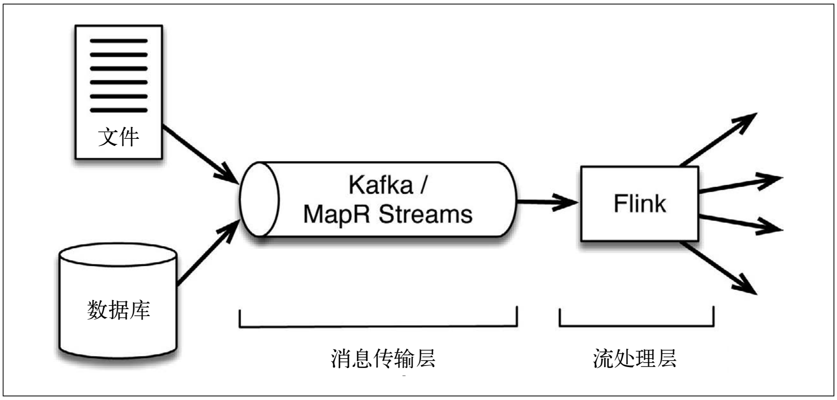



图 2-1：Flink 项目的架构有两个主要组成部分：消息传输层和由 Flink 提供的流处理层。消息传输层负责传输连续事件产生的消息，能够提供消息传输的系统包括 Kafka 和 MapR Streams。MapR Streams 是 MapR 融合数据平台的一个主要组成部分，它兼容 Kafka API

围绕着实时应用程序产生的兴奋感会将人们的注意力集中到流处理层上，并促使人们思考如何根据具体的项目需求选择流处理技术。除了 Flink 之外，还有其他的流处理工具可供选择（比如 Spark Streaming、Storm、Samza 和 Apex）。本书中的示例都以 Flink 作为流处理器。

事实上，在设计高效的流处理架构时，不仅流处理器的选择会造成架构的巨大差异，消息传输层也很关键。现代系统之所以更容易处理大规模的流数据，其中很大一部分原因就是消息传输方式的改进，以及流处理器与消息传输系统的交互方式的改变。

消息传输层需要具备一些特定的功能。目前来看，有两种技术可以很好地提供所需的功能，它们便是 Kafka 和 MapR Streams。MapR Streams 是 MapR 融合数据平台的一部分，它支持 Kafka API。本书中的示例都假设消息传输层采用 Kafka 或 MapR Streams。

## 2.3 消息传输层的理想功能

流处理架构的消息传输层需要具备哪些功能呢？

### 2.3.1 兼具高性能和持久性

消息传输层的一个作用是作为流处理层上游的安全队列——它相当于缓冲区，可以将事件数据作为短期数据保留起来，以防数据处理过程发生中断。直到最近几年，高性能和持久性不可兼得的困境才被打破。人们习惯上认为流数据从消息传输层到流处理层之后就被丢弃：用了就没了。

为了设计新一代的流处理架构，高性能和持久性不可兼得是首先要改变的一个观念。兼具高性能和持久性对于消息传输系统来说至关重要；Kafka 和 MapR Streams 都可以满足这个需求。

具有持久性的好处之一是*消息可以重播*。这个功能使得像 Flink 这样的处理器能对事件流中的某一部分进行重播和再计算（第 5 章会详细介绍）。正是由于消息传输层和流处理层相互作用，才使得像 Flink 这样的系统有了准确处理和“时空穿梭”（指重新处理数据的能力）的保障，认识到这一点至关重要。

### 2.3.2 将生产者和消费者解耦

采用高效的消息传输技术，可以从多个源（生产者）收集数据，并使这些数据可供多个服务或应用程序（消费者）使用，如图 2-2 所示。Kafka 和 MapR Streams 把从生产者获得的数据分配给既定的主题。数据源将数据推送给消息队列，消费者（或消费者群组）则拉取数据。事件数据只能基于给定的偏移量从消息队列中按顺序读出。生产者并不向所有消费者自动广播。这一点听起来微不足道，但是对整个架构的工作方式有着巨大的影响。

这种传输方式——消费者订阅感兴趣的主题——意味着消息立刻到达，*但并不需要被立刻处理*。在消息到达时，消费者并不需要处于运行状态，而是可以根据自身的需求在任何时间使用数据。这样一来，添加新的消费者和生产者也很容易。采用解耦的消息传输系统很有意义，因为它能支持*微服务*，也支持将处理步骤中的实现过程隐藏起来，从而允许自由地修改实现过程。

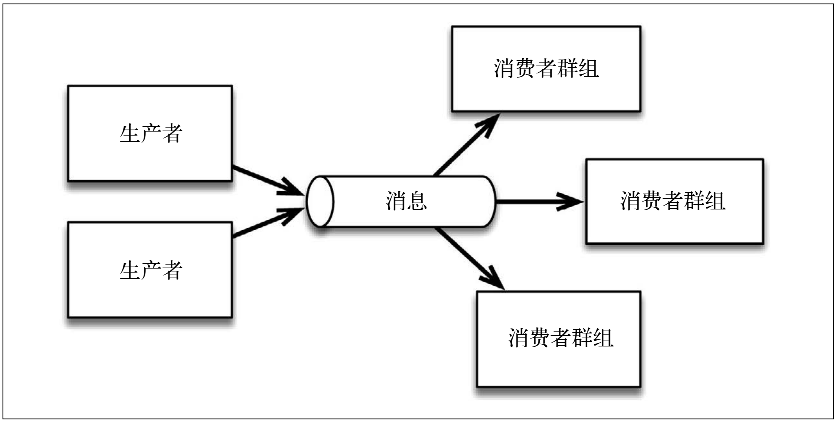



图 2-2：在 Kafka 和 MapR Streams 这样的消息传输工具中，数据的生产者和消费者（Flink 应用程序也是其中的一种消费者）是解耦的。到达的消息既可以立刻被使用，也可以稍后被使用。消费者从队列中订阅消息，而不是由生产者向所有消费者广播。在消息到达的时候，消费者不必处于运行状态

## 2.4　支持微服务架构的流数据

微服务方法指的是将大型系统的功能分割成通常具有单一目的的简单服务，从而使小型团队可以轻松地构建和维护这些服务。即使是超大型组织，也可以用这种设计实现敏捷。若要使整个系统正常工作，各服务之间因通信而产生的连接必须是轻量级的。

> “（微服务的）目标是给予每个团队一项工作，并授以完成工作的方法，然后就放手。”
>
> 摘自 *Streaming Architecture* 一书的第 3 章，该书由 O’Reilly Media 于 2016 年出版，作者是 Ted Dunning 和 Ellen Friedman。

消息传输系统一方面将生产者和消费者解耦，另一方面又有足够高的吞吐量，并且能够满足像 Flink 这样的高性能流处理器。这种系统非常适合用于构建微服务。就连接微服务而言，流数据是相对较新的模式，但是它有许多好处，接下来的几小节将进行解释。

## 2.4.1 数据流作为中心数据源

通过上文的讨论可以看出，流处理架构的核心是使各种应用程序互连在一起的消息队列。流处理器（例如本书介绍的 Flink）从消息队列中订阅数据并加以处理。处理后的数据可以流向另一个消息队列。这样一来，其他应用程序（包括其他 Flink 应用程序）都可以共享流数据。在一些情况下，处理后的数据会被存放在本地数据库中，如图 2-3 所示。

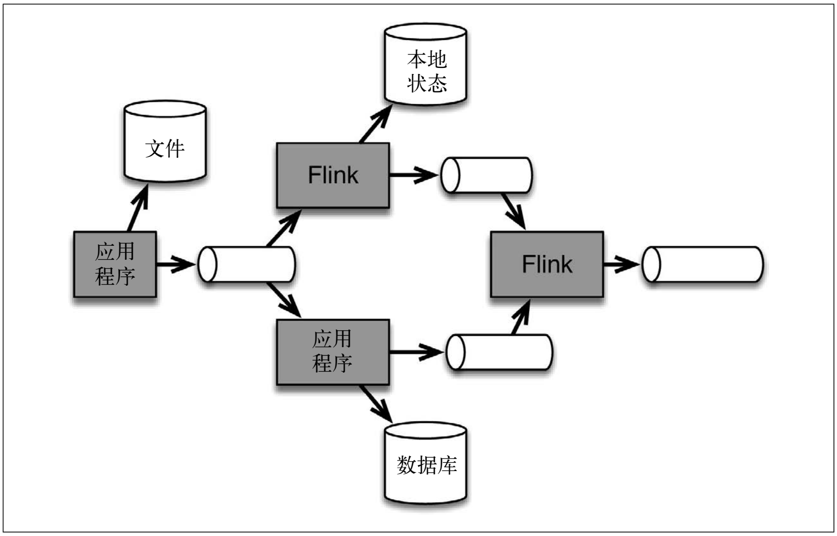



图 2-3：在流处理架构中，消息队列（图中以水平圆柱体表示）连接应用程序，并作为新的共享数据源；它们取代了从前的大型集中式数据库。在本例中，Flink 被多个应用程序使用。本地化的数据能够根据微服务项目的需要被存储在文件或者数据库中。这种流处理架构的另一个好处是，流处理器（例如 Flink）还可以保障数据一致性

流处理架构不需要集中式数据库。取而代之的是消息队列，它作为共享数据源，服务于各种不同的消费者。

## 2.4.2 欺诈检测：流处理架构用例

基于流处理的微服务架构有着强大的灵活性，特别是当同一份数据被用于

不同的场景时，其灵活性更为明显。以信用卡服务提供商的欺诈检测项目为例。项目的目标是尽可能快地识别可疑的刷卡行为，从而阻止盗刷，并将损失降到最低。例如，欺诈检测器可以将刷卡速度作为评判依据：在很短的时间内，发生在跨度很大的不同地点，这样的连续交易合理吗？事实上，真正的欺诈检测器会使用几十个（甚至几百个）特征作为评判依据，但是我们仅仅通过分析刷卡速度这一个特征就可以理解许多问题。

图 2-4 展示了流处理架构在欺诈检测项目中的优势。在图中，许多销售终端（POS 机 1~*n*）请求欺诈检测器判定是否有欺诈行为。这些来自销售终端的询问需要立即被应答，因此在销售终端与欺诈检测器之间形成了询问与应答的交互。

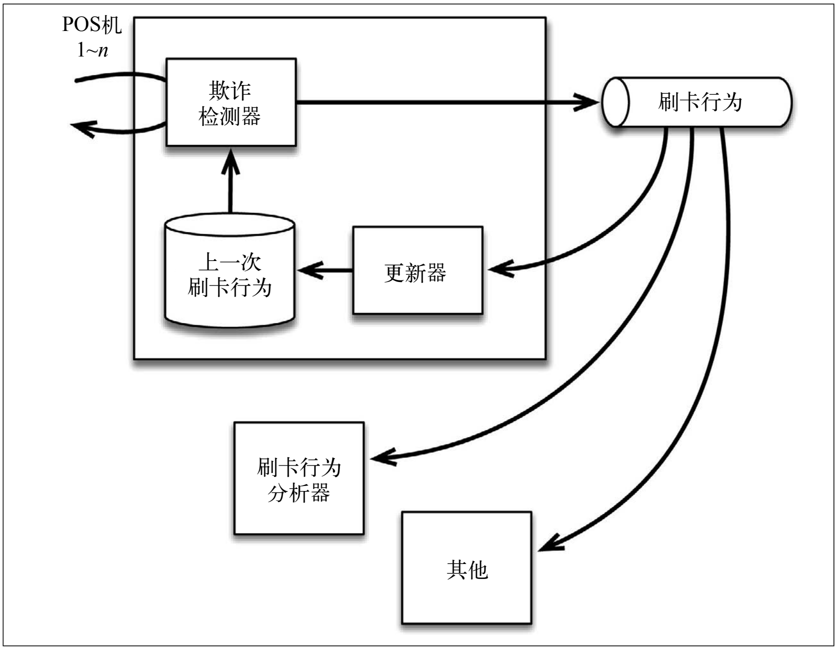



图 2-4：欺诈检测可以从基于流处理的微服务架构中受益。在这种架构中，Flink 在如下几个部分都非常有用：欺诈检测器、更新器，甚至刷卡行为分析器。值得一提的是，这种架构没有直接在本地数据库中更新刷卡行为的数据，而是将数据放在消息队列里，这些数据还可以不受干扰地被刷卡行为分析器等其他服务使用（图片来源：*Streaming Architecture* 第 6 章）

传统的欺诈检测模型将包含每张信用卡最后一次刷卡地点的文件直接存储在数据库中。但在这样的集中式数据库设计中，其他消费者并不能轻易使用刷卡行为的数据，因为访问数据库可能会影响欺诈检测系统的正常工作；在没有经过认真仔细的审查之前，其他消费者绝不会被授权更改数据库。这将导致整个流程变慢，因为必须仔细执行各种检查，以避免核心的业务功能受到破坏或影响。

与传统方法相比，图 2-4 所示的流处理架构设计将欺诈检测器的输出发送给外部的消息队列（Kafka 或 MapR Streams），再由如 Flink 这样的流处理器更新数据库，而不是直接将输出发送给数据库。这使得刷卡行为的数据可以通过消息队列被其他服务使用，例如刷卡行为分析器。上一次刷卡行为的数据被存储在本地数据库中，不会被其他服务访问。这样的设计避免了因为增加新的服务而带来的过载风险。

### 2.4.3 给开发人员带来的灵活性

基于流处理的微服务架构也为欺诈检测系统的开发人员带来了灵活性。假设开发团队正试图改进欺诈检测模型并加以评估。刷卡行为产生的消息流可以被新模型采用，而完全不影响已有的检测器。新增加一个数据消费者的开销几乎可以忽略不计，同时只要合适，数据的历史信息可以保存成任何一种格式，并且使用任意的数据库服务。此外，如果刷卡行为队列中的消息被设计成业务级别的事件，而不是数据库表格的更新，那么消息的形式和内容都会非常稳定。若一定要更改，向前兼容可以避免更改已有的应用程序。

信用卡欺诈检测只是流处理架构的一个用例。流处理架构通过一个合适的消息传输系统（Kafka 或 MapR Streams）和一个多用途、高性能的流处理器（Flink），能支持各种应用程序使用共享数据源，即消息流。

## 2.5 不限于实时应用程序

虽然低延迟性很重要，但是实时应用程序只是众多流数据消费者中的一种。流数据的应用很广泛，比如，流处理应用程序可以通过订阅消息队列中的流数据来实时更新仪表盘（如图 2-5 中的 A 组消费者）。

持久化的消息可以被重播，这一特性使许多用户获益（如图 2-5 中的 C 组消费者）。在本例中，消息流成为了可审计的日志，或者长期的事件历史。能够重现的历史非常有用，比如可以在工业安全分析中作为预知维护模型的一部分输入，也可以在医学或环境科学领域用于回顾性研究。

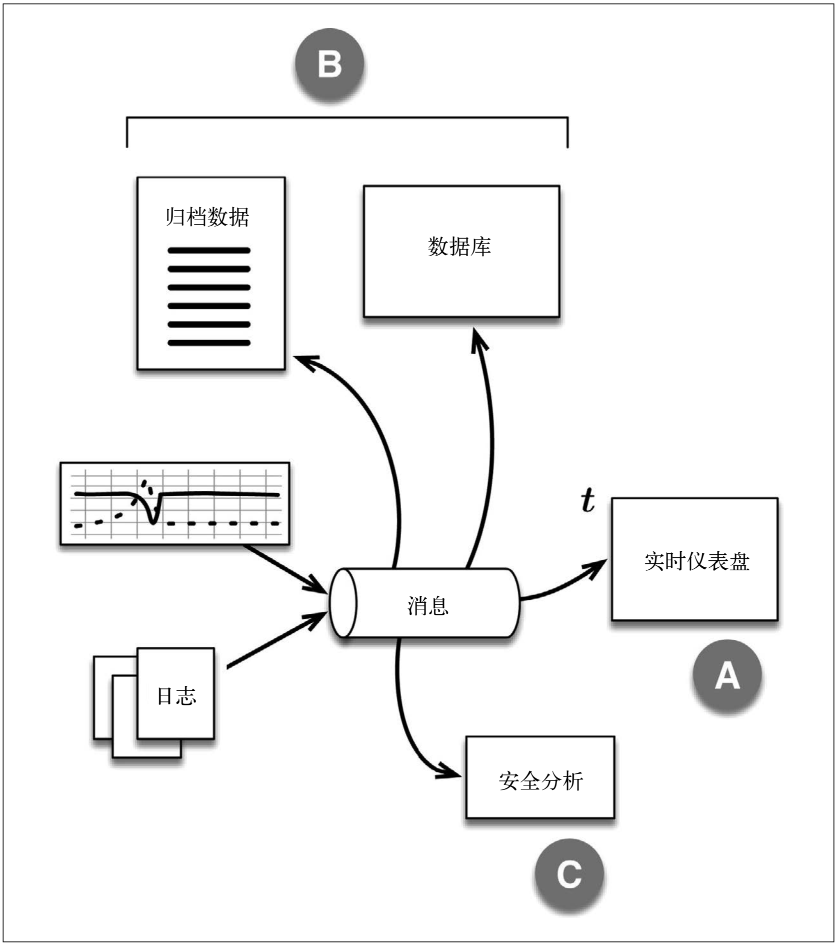



图 2-5：流数据消费者并不仅限于实时应用程序，尽管它们是很重要的一种。本图展示了从流处理架构中获益的几类消费者。A 组消费者可能做各种实时分析，包括实时更新仪表盘。B 组消费者记录数据的当前状态，这些数据可能同时也被存储在数据库或搜索文件中

在其他一些用例中，应用程序使用消息队列中的数据更新本地数据库或者搜索文件（如图 2-5 中的 B 组消费者）。消息队列中的数据往往必须被流处理器聚合或者分析并转换之后，才会输出到数据库中。这是 Flink 擅长的另一个场景。

## 2.6 流的跨地域复制

流处理架构并不是玩具：它被用于具有重要使命的系统，这些系统要求流处理层和消息传输层具备一些特性。许多关键的业务系统依靠跨数据中心的一致性，它们不仅需要高效的流处理层，更需要消息传输层拥有可靠的跨地域复制能力。例如，电信公司需要在移动通信基站、用户和处理中心之间共享数据；金融机构需要快速、准确地在相隔很远的办公室之间复制数据，同时控制成本。类似的例子还有很多。

具体来说，数据中心之间的数据复制需要保存消息偏移量，这一点最有用，因为它使得任何数据中心的更新都可以被传播到其他数据中心，且允许双向和循环的数据复制。如果消息偏移量没有被保存，那么另一个数据中心就无法可靠地重启程序。如果不允许其他数据中心更新数据，那么就必须设计可靠的主节点。循环复制则可以避免复制过程出现单点故障。

MapR Streams 目前支持这些功能，但 Kafka 还不支持[1]。MapR Streams 的基本原理是，许多流主题被收集在一级数据结构中，该结构也叫作流，它与文件、表格和目录共存于 MapR 的数据平台。这些流成为管理数据复制、生存时间和访问权限的基础。在流中针对主题所做的变更将被贴上源集群的 ID，避免它被无限地循环复制。之后，这些变更被依次传播到其他集群中，并且保留所有的消息偏移量。

跨数据中心的流复制能力扩展了流数据和流处理的用途。以在线广告业务为例，流数据分析可以从多个方面提供帮助。结合图 2-5 中的流数据消费者分类，在广告技术领域，实时应用程序（A 组消费者）可能面临实时的广告资源管理，数据库（B 组消费者）当前状态的视图可能是 cookie 档案，流重播（C 组消费者）则可以用来检测虚假点击。

注 1：作者在此处描述的是 2016 年的情况。至于 Kafka 目前是否支持，请查阅 Kafka 官方文档。——编者注

一个挑战是，同一个广告的不同竞价由不同的数据中心处理，但数据都取自同一个广告资源池。在准确性和速度都非常关键的广告技术领域，不同的数据中心如何协调可用的广告资源呢？消息流作为正确的数据源被各个数据中心共享。在本例中，跨数据中心的流复制能力非常有用，MapR Streams 就有这样的能力。

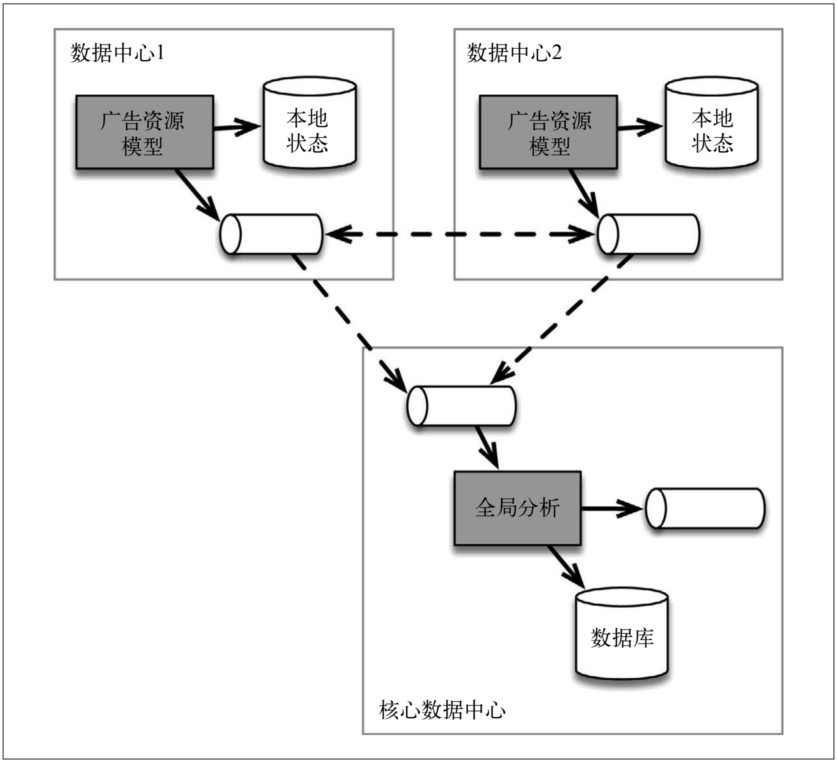



图 2-6：来自广告技术领域的例子：流数据分析在不同的数据中心进行，且数据中心有多种基于模型的应用程序，Flink 对这些应用程序非常有用。每个本地数据中心都需要保存自己的当前事务状态，而它们都从同一个广告资源池读取数据。另一个需求是与核心数据中心共享数据，并且可以利用 Flink 进行全局分析。本用例需要高效且准确地进行跨地域复制

除了能让所有的业务部门了解共享资源的最新状态（这个例子在其他许多行业也同样适用），跨数据中心的流复制能力还有其他优势。拥有不止一个

数据中心，可以缓解高吞吐所带来的压力；广告的竞价和投放可以由靠近最终用户的数据中心处理，从而降低网络传输带来的延迟。另外，多个数据中心也可以进行备份，从而防止突发灾难时丢失数据。

我们在第 1 章和第 2 章中看到，流处理的方式符合连续事件发生的自然规律。我们还讨论了流处理架构的诸多好处，这种架构结合了 Kafka 或 MapR Streams 等高效的消息传输技术，以及 Flink 流处理器。

在第 3 章中，我们将探讨 Flink 的关键特性，并在深入学习 Flink 的工作原理之前，概览它的用途。

第 3 章

# Flink的用途

Flink 为流处理器开辟了新的用武之地，它使第 2 章所述的流处理架构变得完整。它的一大优势便是，使应用程序的构建过程符合自然规律。为了了解 Flink 的用途及用法，我们来看一看令它具有多用途的几个核心特点，特别是它如何保障数据的正确性。

## 3.1 不同类型的正确性

第 1 章介绍了流处理欠佳的后果。本章讨论 Flink 如何正确地进行流处理，以及保障正确性到底意味着什么。人们往往将正确性等同于准确性——以计数为例，计数结果是否“正确”？这个例子很好地诠释了正确性，但是正确性的影响因素很多，当思考计算怎样才能契合需要建模和分析的场景时，更是如此。换句话说，在处理数据时，需要解决这几个问题：“我需要什么？”“我期望什么？”“我在什么时候需要得到结果？”

### 3.1.1 符合产生数据的自然规律

流处理器（尤其是 Flink）的正确性体现在计算窗口的定义符合数据产生的自然规律。举个例子，通过点击流数据追踪某网站的 3 个访问者（图 3-1 中的 A、B 和 C）的活动。对于每个访问者来说，活动是不连续的。在访问时

间段内，事件数据被收集起来；当访问者起身去喝茶或喝咖啡时，或者当他们因为老板从身边经过而切换回工作页面时，数据就产生了间隙。处理框架能够将访问者行为分析的计算窗口与实际的访问时间段吻合到什么程度？换句话说，计算窗口与会话窗口吻合吗？

首先让我们来看看，当访问者行为分析通过微批处理方法或者固定的计算窗口来处理时，会发生什么情况，如图 3-1 所示。

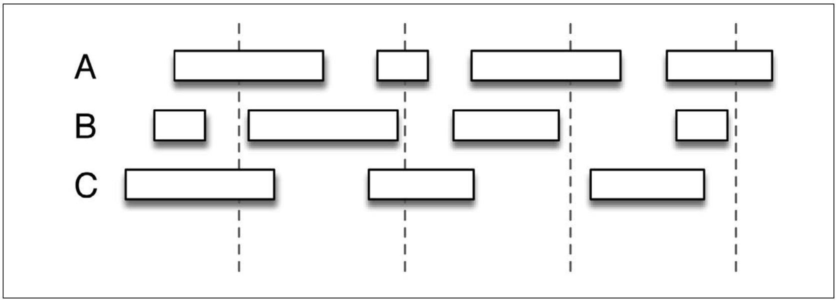



图 3-1：采用微批处理方法时，很难使计算窗口（虚线所示）与会话窗口（长方形所示）吻合

由微批处理方法得到的计算窗口是人为设置的，因此很难与会话窗口吻合。使用 Flink 的流处理 API，可以更灵活地定义计算窗口，因此这个问题迎刃而解。举个例子，开发人员可以设置非活动阈值，若超过这个阈值（例如 5 分钟），就可以判断活动结束。图 3-2 展示了这种开窗方式。

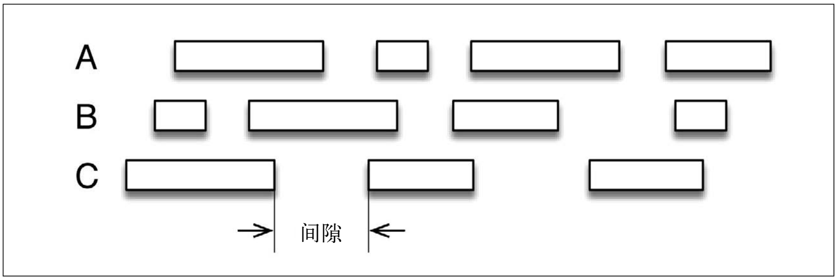



图 3-2：Flink 的流处理能力能够使计算窗口与会话窗口吻合。如图所示，计算窗口随间隙出现。在本例中，相邻事件之间都有间隙，间隙的时长都超过了预先定义的阈值

Flink 能做到这一点的根本原因是，它可以根据真实情况设置计算窗口。

> **事件时间与处理时间**
>
> 值得注意的是，时间的指定方式不止一种。在图 3-2 的例子中，为了将事件指定给某一个窗口，程序员很有可能会选择采用事件时间，即事件实际发生的时间。另一种方式是采用处理时间，即事件流数据开始被程序处理的时间。第 4 章将深入介绍 Flink 中的时间概念以及开窗方式。

## 3.1.2 事件时间

一般而言，流处理架构不常采用事件时间，尽管越来越多的人这样做。Flink 能够完美地做到这一点，这在实现计算的正确性上非常有用。为了获得最佳的计算结果，系统需要能够通过数据找到事件发生的时间，而不是只采用处理时间。

Flink 理解事件时间的这种能力保障了正确性。2016 年，时任 data Artisans 公司应用工程总监的 Jamie Grier 在 OSCON 大会上展示了这一点。他通过生成的数据模拟压力传感器的测量结果，并写了一个 Flink 程序来计算以 1 秒为计算窗口、每秒内正弦波的数值之和。正确的结果是 0。他比较了用处理时间划分窗口和用事件时间划分窗口的差别。采用处理时间时，结果总是或多或少地有些偏差；采用事件时间时，则总是可以获得正确的结果，如图 3-3 所示。

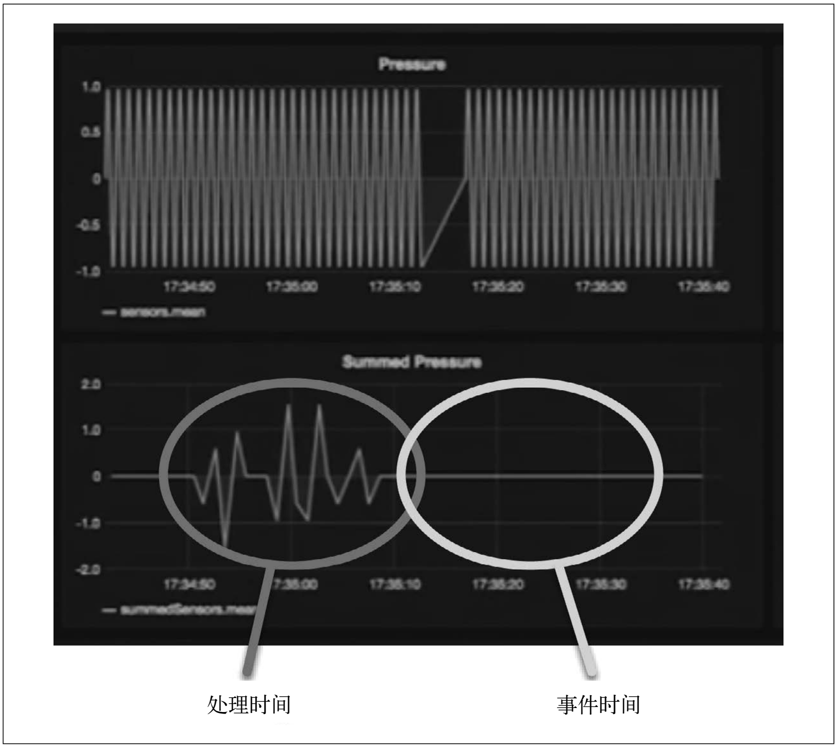



图 3-3：从处理时间切换到事件时间，让许多计算工作完成得更好。用处理时间来计算会导致错误，用事件时间则能得到正确的结果（图片来源：Jamie Grier 于 2016 年 5 月在 OSCON 大会上所做的演示）

与其他流处理系统相比，Flink 的一个优势就是能区分不同类型的时间。

### 3.1.3 发生故障后仍保持准确

若想使计算保持准确，就必须跟踪计算状态。如果计算框架本身不能做到这一点，就必须由应用程序的开发人员来完成这个任务。连续的流处理很难跟踪计算状态，因为计算过程没有终点。实际上，对状态的更新是持续进行的。

Flink 解决了可能影响正确性的几个问题，包括如何在故障发生之后仍能进行有状态的计算。

Flink 所用的技术叫作检查点（checkpoint），第 5 章会详细介绍它的原理。在每个检查点，系统都会记录中间计算状态，从而在故障发生时准确地重置。这一方法使系统以低开销的方式拥有了容错能力——当一切正常时，检查点机制对系统的影响非常小。

值得注意的是，检查点也是 Flink 能够按需重新处理数据的关键所在。毕竟，并不是只有在发生故障之后才会重新处理数据。比如，在运行新模型或者修复 bug 时，就可能需要重播并重新处理事件流数据。Flink 成全了这些操作。

Flink 的检查点特性在流处理器中是独一无二的，它使得 Flink 可以准确地维持状态，并且高效地重新处理数据。

## 3.1.4 及时给出所需结果

Flink 能够满足低延迟应用程序的需要，将这算作一种正确性可能出人意料。我们换个角度看看：有些计算结果或许很准确，例如求和或者求平均值的结果，但是如果没有及时地取得结果，那么很难说它们是正确的。举一个例子，假设你在开车上班的途中想通过智能手机上的实时路况查询及导航应用程序选择一条畅通的路，如果应用程序花了 2 小时才把查询结果发给你，那么结果再准确也是无用的。哪怕只有 5 秒钟的延迟也足以造成麻烦，因为你可能已经拐错了弯。

可见，在某些情况下，极低的延迟非常重要，它决定了系统能够及时地给出所需结果，而不仅仅是完成计算。Flink 的实时且容错的流处理能力可以满足这类需求。

## 3.1.5 使开发和运维更轻松

Flink 与用户交互的接口也有助于保障正确性。完备的语义简化了开发工作，进而降低了出错率。此外，Flink 还承担了跟踪计算状态的任务，从而减轻了开发人员的负担，简化了编程工作，并提高了应用程序的成功率。用同一种技术来实现流处理和批处理，大大地简化了开发和运维工作。

## 3.2 分阶段采用Flink

尽管 Flink 拥有非常丰富的功能，并能处理极为复杂的数据，但是没有必要为了采用 Flink 而彻底抛弃其他技术。流处理架构可以分步来实现。有些公司在引入流处理架构时，先实现简单的应用程序，等到熟悉后再推广。虽然试点应用程序的类型完全取决于公司的需求，但是许多公司都有相似的流处理“价值链”。

接下来，让我们抓住机会深入了解 Flink 能做什么以及它是如何做到的。第 4~6 章将介绍 Flink 的一些基本功能。

# 第 4 章

# 对时间的处理

用流处理器编程和用批处理器编程最关键的区别在于对时间的处理。举一个非常简单的例子：计数。事件流数据（如微博内容、点击数据和交易数据）不断产生，我们需要用 key 将事件分组，并且每隔一段时间（比如一小时）就针对每一个 key 对应的事件计数。这是众所周知的“大数据”应用，与 MapReduce 的词频统计例子相似。

## 4.1 采用批处理架构和Lambda架构计数

尽管看起来简单，但是大规模的计数任务在实践中出人意料地困难。当然，计数无处不在。针对联机分析处理多维数据集的聚合或其他操作，都可以简单地归结为计数。图 4-1 展示了如何采用传统的批处理架构实现计数任务。

在该架构中，持续摄取数据的管道每小时创建一次文件。这些文件通常被存储在 HDFS 或 MapR-FS 等分布式文件系统中。像 Apache Flume 这样的工具可以用于完成上述工作。由调度程序安排批处理作业（如 MapReduce 作业）分析最近生成的一个文件（将文件中的事件按 key 分组，计算每个 key 对应的事件数），然后输出计数结果。对于每个使用 Hadoop 的公司来说，其集群都有多个类似的管道。

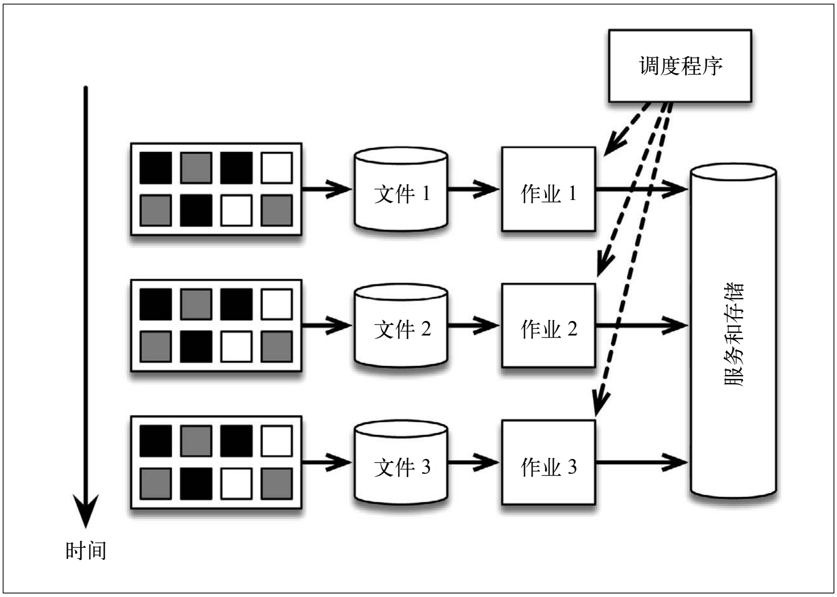



图 4-1：用定期运行的批处理作业来实现应用程序的持续性。数据被持续地分割为文件（如以一小时为单位）；然后，批处理作业将文件作为输入，以此达到持续处理数据的效果

这种架构完全可行，但是存在以下问题。

- 太多独立的部分。为了计算数据中的事件数，这种架构动用了太多系统。每一个系统都有学习成本和管理成本，还可能存在 bug。
- 对时间的处理方法不明确。假设需要改为每 30 分钟计数一次。这个变动涉及工作流调度逻辑（而不是应用程序代码逻辑），从而使 DevOps 问题与业务需求混淆。
- 预警。假设除了每小时计数一次外，还需要尽可能早地收到计数预警（比如在事件数超过 10 时预警）。为了做到这一点，可以在定期运行的批处理作业之外，引入 Storm 来采集消息流（Kafka 或者 MapR Streams）。Storm 实时提供近似的计数，批处理作业每小时提供准确的计数。但是这样一来，就向架构增加了一个系统，以及与之相关的新编程模型。上述架构叫作 Lambda 架构，如图 4-2 所示。第 1 章有过简要的介绍。

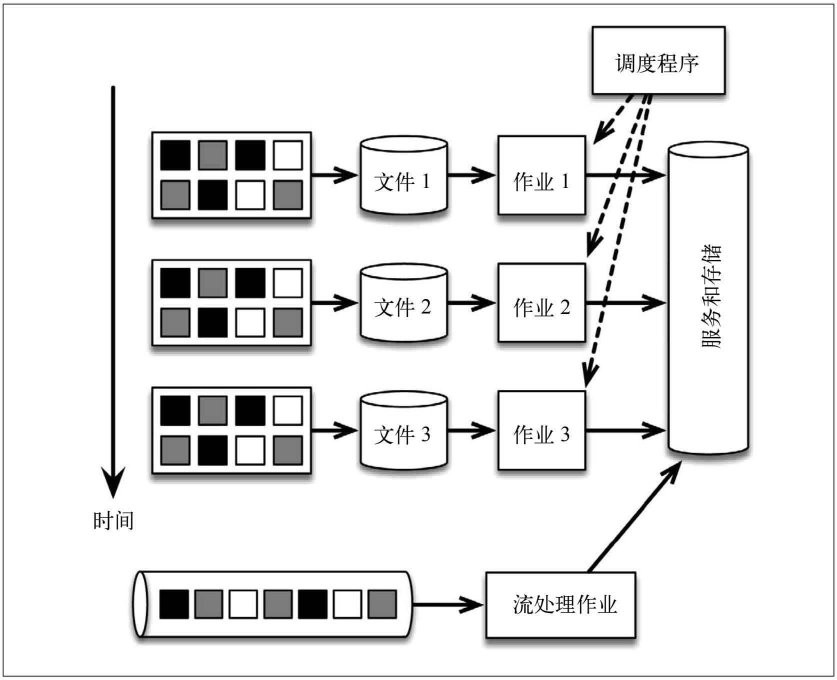



图 4-2：Lambda 架构用定期运行的批处理作业来实现应用程序的持续性，并通过流处理器获得预警。流处理器实时提供近似结果；批处理层最终会对近似结果予以纠正

- *乱序事件流*。在现实世界中，大多数事件流都是乱序的，即事件的实际发生顺序（事件数据在生成时被附上时间戳，如智能手机记录下用户登录应用程序的时间）和数据中心所记录的顺序不一样。这意味着本属于前一批的事件可能被错误地归入当前一批。批处理架构很难解决这个问题，大部分人则选择忽视它。
- *批处理作业的界限不清晰*。在该架构中，“每小时”的定义含糊不清，分割时间点实际上取决于不同系统之间的交互。充其量也只能做到大约每小时分割一次，而在分割时间点前后的事件既可能被归入前一批，也可能被归入当前一批。将数据流以小时为单位进行分割，实际上是最简单的方法。假设需要根据产生数据的时间段（如从用户登录到退出）生成聚合结果，而不是简单地以小时为单位分割数据，则用如图 4-1 和图 4-2 所示的架构无法直接满足需求。

## 4.2 采用流处理架构计数

当然有更好的办法来对事件流进行计数。如你所想，计数是流处理用例，上一节只是使用定期运行的批处理作业来模拟流处理。此外，必须把各种系统耦合在一起。图 4-3 展示了采用流处理架构的应用程序模型。

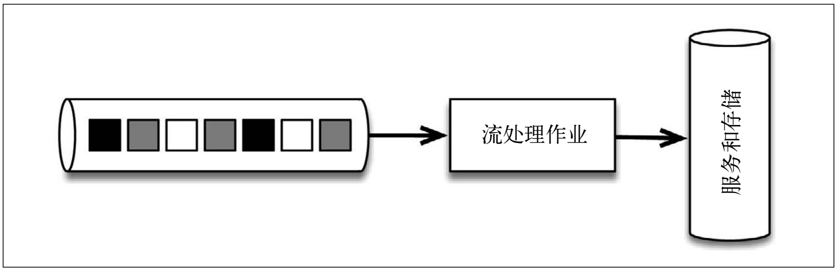



图 4-3：通过流处理架构实现应用程序的持续性。水平圆柱体表示消息传输系统（Kafka 或 MapR Streams）。消息传输系统为负责处理所有数据的流处理器（在本例中是 Flink）提供流数据，产生的结果既是实时的，也是正确的

事件流由消息传输系统提供，并且只被单一的 Flink 作业处理，从而以小时为单位计数和预警（后者可选）。这种方法直接解决了上一节提到的所有问题。Flink 作业的速度减慢或者吞吐量激增只会导致事件在消息传输系统中堆积。以时间为单位把事件流分割为一批批任务（称作窗口），这种逻辑完全嵌入在 Flink 程序的应用逻辑中。预警由同一个程序生成，乱序事件由 Flink 自行处理。要从以固定时间分组改为根据产生数据的时间段分组，只需在 Flink 程序中修改对窗口的定义即可。此外，如果应用程序的代码有过改动，只需重播 Kafka 主题，即可重播应用程序。采用流处理架构，可以大幅减少需要学习、管理和编写代码的系统。Flink 应用程序用来计数的代码非常简单，如下所示。

```
DataStream<LogEvent> stream = env
  // 通过Kafka生成数据流
  .addSource(new FlinkKafkaConsumer(...))
  // 分组
  .keyBy("country")
  // 将时间窗口设为60分钟
  .timeWindow(Time.minutes(60))
  // 针对每个时间窗口进行操作
  .apply(new CountPerWindowFunction());
```

流处理区别于批处理最主要的两点是：流即是流，不必人为地将它分割为文件；时间的定义被明确地写入应用程序代码（如以上代码的时间窗口），而不是与摄取、计算和调度等过程牵扯不清。

## 流处理系统中的批处理

第 1 章讨论过微批处理，它是介于流处理和批处理之间的方法。实际上，微批处理的含义取决于具体情况。从某种角度来说，图 4-1 中的批处理架构也可以称为微批处理架构，前提是文件足够小。

Storm Trident 是这样实现微批处理的：它先创建一个大的 Storm 事件，其中包含固定数量的子事件；然后将这些聚合在一起的子事件用持续运行的 Storm 拓扑处理。Spark Streaming 的微批处理架构和批处理架构本质上是一致的，只不过对用户隐藏了前两步（摄取和存储），并将每份微批数据用预写日志（而不是文件）的形式存储在内存中。包括 Flink 在内的所有现代流处理器在内部都使用了某种形式的微批处理技术，在 shuffle 阶段将含有多个事件的缓冲容器通过网络发送，而不是发送单个事件。这些形式各不相同。

需要说明的是，流处理系统中的批处理必须符合以下两点要求。

- 批处理只作为提高系统性能的机制。批量越大，系统的吞吐量就越大。
- 为了提高性能而使用的批处理必须完全独立于定义窗口时所用的缓冲，或者为了保证容错性而提交的代码，也不能作为 API 的一部分。否则，系统将受到限制，并且变得脆弱且难以使用。

应用程序开发人员和数据处理系统的用户不需要考虑系统是否可以执行微批处理以及如何执行，而是需要考虑系统是否可以处理乱序事件流以及不一致的窗口，是否可以在提供准确的聚合结果之外还提供预警，以及是否可以准确地重播历史数据；还需要考虑系统的性能特征（低延迟和高吞吐），以及如何在发生故障时保证系统持续运行。

# 4.3 时间概念

在流处理中，主要有两个时间概念[1]。

- 事件时间，即事件实际发生的时间。更准确地说，每一个事件都有一个与它相关的时间戳，并且时间戳是数据记录的一部分（比如手机或者服务器的记录）。事件时间其实就是时间戳。
- 处理时间，即事件被处理的时间。处理时间其实就是处理事件的机器所测量的时间。

图 4-4 说明了事件时间和处理时间的区别。

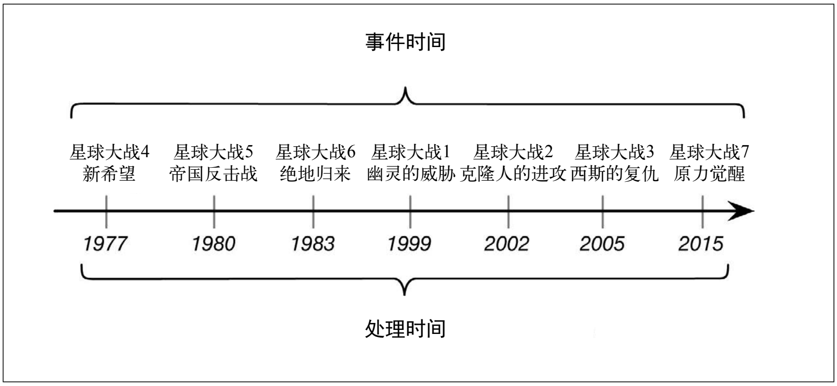



图 4-4：事件时间顺序与处理时间顺序不一致的乱序事件流

以《星球大战》系列电影为例。首先上映的 3 部电影是该系列中的第 4、5、6 部（这是事件时间），它们的上映年份分别是 1977 年、1980 年和 1983 年（这是处理时间）。之后按事件时间上映的第 1、2、3、7 部，对应的处理时间分别是 1999 年、2002 年、2005 年和 2015 年。由此可见，事件流的顺序可能是乱的（尽管年份顺序一般不会乱）。

注 1：本章涉及的许多想法都是由 Google Dataflow 团队（即现在的 Apache Beam 团队）提出的。团队成员包括 Tyler Akidau 和 Frances Perry 等人。如果想了解更多关于 Dataflow 模型的内容，请参考 Tyler Akidau 的文章 *Streaming 101* 和 *Streaming 102*。Flink 处理时间和窗口的机制很大程度上来源于 Tyler Akidau 等人所著的关于 Dataflow 模型的论文。

通常还有第 3 个时间概念，即*摄取时间*，也叫作*进入时间*。它指的是事件进入流处理框架的时间。缺乏真实事件时间的数据会被流处理器附上时间戳，即流处理器第一次看到它的时间（这个操作由 source 函数完成，它是程序的第一个处理节点）。

在现实世界中，许多因素（如连接暂时中断，不同原因导致的网络延迟，分布式系统中的时钟不同步，数据速率陡增，物理原因，或者运气差）使得事件时间和处理时间存在偏差（即*事件时间偏差*）。事件时间顺序和处理时间顺序通常不一致，这意味着事件以乱序到达流处理器。

*根据应用程序的不同，两个时间概念都很有用*。有些应用程序（如一些预警应用程序）需要尽可能快地得到结果，即使有小的误差也没关系。它们不必等待迟到的事件，因此适合采用处理时间语义。其他一些应用程序（如欺诈检测系统或者账单系统）则对准确性有要求：只有在时间窗口内发生的事件才能被算进来。对于这些应用程序来说，事件时间语义才是正确的选择。也有两者都采用的情况，比如既要准确地计数，又要提供异常预警。

> Flink 允许用户根据所需的语义和对准确性的要求选择采用事件时间、处理时间或摄取时间定义窗口。

当采用事件时间定义窗口时，应用程序可以处理乱序事件流以及变化的事件时间偏差，并根据事件实际发生的时间计算出有意义的结果。

# 4.4 窗口

4.1 节举例说明了如何在 Flink 中定义时间窗口并以小时为单位生成聚合结果。*窗口*是一种机制，它用于将许多事件按照时间或者其他特征分组，从而将每一组作为整体进行分析（比如求和）。

## 4.4.1 时间窗口

*时间窗口*是最简单和最有用的一种窗口。它支持*滚动*和*滑动*。举一个例子，假设要对传感器输出的数值求和。

一分钟滚动窗口收集最近一分钟的数值，并在一分钟结束时输出总和，如图 4-5 所示。

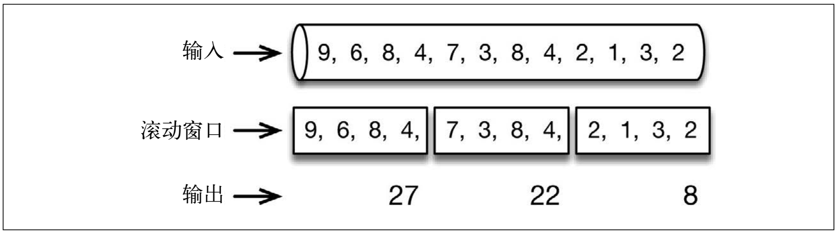



图 4-5：一分钟滚动窗口计算最近一分钟的数值总和

一分钟滑动窗口计算最近一分钟的数值总和，但每半分钟滑动一次并输出结果，如图 4-6 所示。

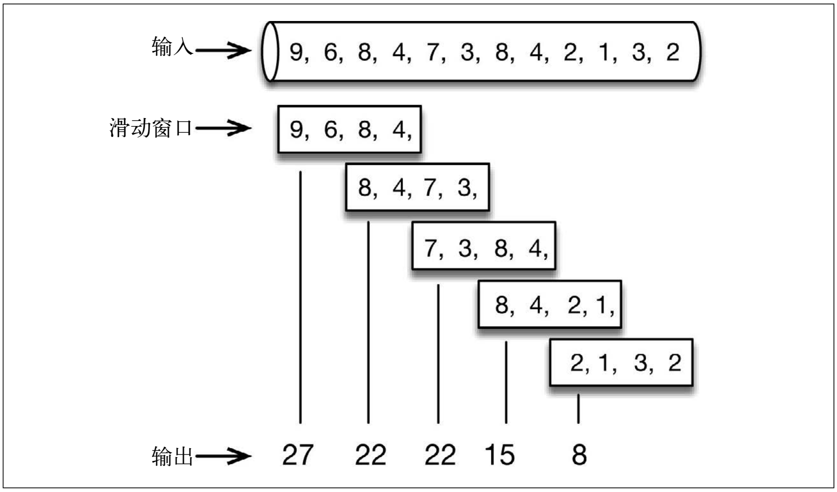



图 4-6：一分钟滑动窗口每半分钟计算一次最近一分钟的数值总和

第一个滑动窗口对 9、6、8 和 4 求和，得到 27。半分钟后，窗口滑动，然后对 8、4、7 和 3 求和，得到 22，照此类推。

在 Flink 中，一分钟滚动窗口的定义如下。

```
stream.timeWindow(Time.minutes(1))
```

每半分钟（即 30 秒）滑动一次的一分钟滑动窗口如下所示。

```
stream.timeWindow(Time.minutes(1), Time.seconds(30))
```

### 4.4.2 计数窗口

Flink 支持的另一种常见窗口叫作*计数窗口*。采用计数窗口时，分组依据不再是时间戳，而是元素的数量。例如，图 4-6 中的滑动窗口也可以解释为由 4 个元素组成的计数窗口，并且每两个元素滑动一次。滚动和滑动的计数窗口分别定义如下。

```
stream.countWindow(4)
stream.countWindow(4, 2)
```

虽然计数窗口有用，但是其定义不如时间窗口严谨，因此要谨慎使用。时间不会停止，而且时间窗口总会“关闭”。但就计数窗口而言，假设其定义的元素数量为 100，而某个 key 对应的元素永远达不到 100 个，那么窗口就永远不会关闭，被该窗口占用的内存也就浪费了。一种解决办法是用时间窗口来触发超时，4.4.4 节会详细介绍。

### 4.4.3 会话窗口

Flink 支持的另一种很有用的窗口是*会话窗口*。第 3 章提到过这个概念。会话指的是活动阶段，其前后都是非活动阶段，例如用户与网站进行一系列交互（活动阶段）之后，关闭浏览器或者不再交互（非活动阶段）。会话需要有自己的处理机制，因为它们通常没有固定的持续时间（有些 30 秒就结束了，有些则长达一小时），或者没有固定的交互次数（有些可能是 3 次点击后购买，另一些可能是 40 次点击却没有购买）。

> Flink 是目前唯一[2]支持会话窗口的开源流处理器。

在 Flink 中，会话窗口由超时时间设定，即希望等待多久才认为会话已经结

注 2：作者在此处描述的是 2016 年的情况。目前，Apache Beam 和 Apache Spark 也支持会话窗口。——译者注

束。举例来说，以下代码表示，如果用户处于非活动状态长达 5 分钟，则认为会话结束。

```
stream.window(SessionWindows.withGap(Time.minutes(5))
```

### 4.4.4 触发器

除了窗口之外，Flink 还提供触发机制。触发器控制生成结果的时间，即何时聚合窗口内容并将结果返回给用户。每一个默认窗口都有一个触发器。例如，采用事件时间的时间窗口将在收到水印时被触发。对于用户来说，除了收到水印时生成完整、准确的结果之外，也可以实现自定义的触发器（例如每秒提供一次近似结果）。

### 4.4.5 窗口的实现

在 Flink 内部，所有类型的窗口都由同一种机制实现。虽然实现细节对于普通用户来说并不重要，但是仍然需要注意以下两点。

- 开窗机制与检查点机制（第 5 章将详细讨论）完全分离。这意味着窗口时长不依赖于检查点间隔。事实上，窗口完全可以没有“时长”（比如上文中的计数窗口和会话窗口的例子）。
- 高级用户可以直接用基本的开窗机制定义更复杂的窗口形式（如某种时间窗口，它可以基于计数结果或某一条记录的值生成中间结果）。

## 4.5 时空穿梭

流处理架构的一个核心能力是*时空穿梭*。如果所有的数据处理工作都由流处理器完成，那么应用程序如何演进呢？我们如何处理历史数据，又如何重新处理数据呢？（假设出于调试或者审计的目的，需要重新处理数据。）

如图 4-7 所示，时空穿梭意味着将数据流倒回至过去的某个时间，重新启动处理程序，直到处理至当前时间为止。像 Kafka 和 MapR Streams 这样的现代传输层，支持时空穿梭，这使得它们与更早的解决方案有所区别。实时流处理总是在处理最近的数据（即图中“当前时间”的数据），历史流处理则从过去开始，并且可以一直处理至当前时间。

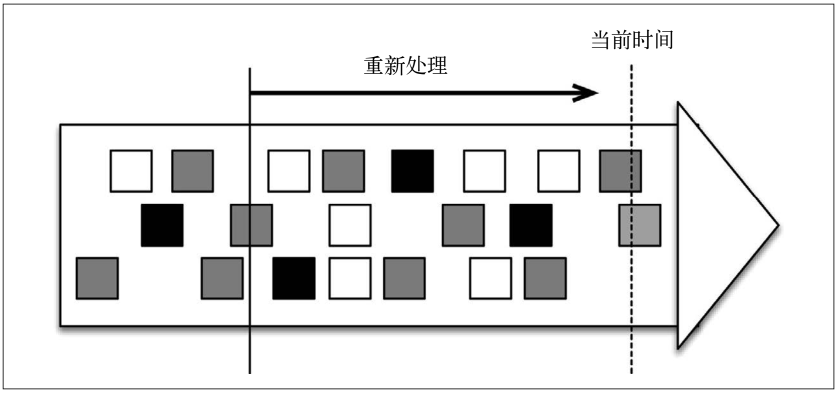



图 4-7：流处理架构拥有时空穿梭（即重新处理数据）的能力。流处理器支持事件时间，这意味着将数据流“倒带”，用同一组数据重新运行同样的程序，会得到相同的结果

> 若要按时间回溯并正确地重新处理数据，流处理器必须支持事件时间。

如果窗口的设定是根据系统时间而不是时间戳，那么每次运行同样的程序，都会得到不同的结果。事件时间使数据处理结果具有确定性，因为用同一组数据运行同样的程序，会得到相同的结果。

## 4.6 水印

支持事件时间对于流处理架构而言至关重要，因为事件时间能保证结果正确，并使流处理架构拥有重新处理数据的能力。当计算基于事件时间时，如何判断所有事件是否都到达，以及何时计算和输出窗口的结果呢？换言之，如何追踪事件时间，并知晓输入数据已经流到某个事件时间了呢？为了追踪事件时间，需要依靠由数据驱动的时钟，而不是系统时钟。

以图 4-5 中的一分钟滚动窗口为例。假设第一个窗口从 10:00:00 开始（即从 10 时 0 分 0 秒开始），需要计算从 10:00:00 到 10:01:00 的数值总和。当时间就是记录的一部分时，我们怎么知道 10:01:00 已到呢？换句话说，我们怎么知道盖有时间戳 10:00:59 的元素还没到呢？

Flink 通过水印来推进事件时间。水印是嵌在流中的常规记录，计算程序通过水印获知某个时间点已到。对于上述一分钟滚动窗口，假设水印标记时间为 10:01:00（或者其他时间，如 10:03:43），那么收到水印的窗口就知道不会再有早于该时间的记录出现，因为所有时间戳小于或等于该时间的事件都已经到达。这时，窗口可以安全地计算并给出结果（总和）。水印使事件时间与处理时间完全无关。迟到的水印（“迟到”是从处理时间的角度而言）并不会影响结果的正确性，而只会影响收到结果的速度。

## 水印是如何生成的

在 Flink 中，水印由应用程序开发人员生成，这通常需要对相应的领域有一定的了解。完美的水印永远不会错：时间戳小于水印标记时间的事件不会再出现。在特殊情况下（例如非乱序事件流），最近一次事件的时间戳就可能是完美的水印。启发式水印则相反，它只估计时间，因此有可能出错，即迟到的事件（其时间戳小于水印标记时间）晚于水印出现。针对启发式水印，Flink 提供了处理迟到元素的机制。

设定水印通常需要用到领域知识。举例来说，如果知道事件的迟到时间不会超过 5 秒，就可以将水印标记时间设为收到的最大时间戳减去 5 秒。另一种做法是，采用一个 Flink 作业监控事件流，学习事件的迟到规律，并以此构建水印生成模型。



水印为以事件时间说明输入数据的完整性提供了一种机制，这种机制可能是启发式的。

如果水印迟到得太久，收到结果的速度可能就会很慢，解决办法是在水印到达之前输出近似结果（Flink 可以实现）。如果水印到达得太早，则可能收到错误结果，不过 Flink 处理迟到数据的机制可以解决这个问题。上述问题看起来很复杂，但是恰恰符合现实世界的规律——大部分真实的事件流都是乱序的，并且通常无法了解它们的乱序程度（因为理论上不能预见未来）。水印是唯一让我们直面乱序事件流并保证正确性的机制；否则只能选择忽视事实，假装错误的结果是正确的。

# 4.7 真实案例：爱立信公司的Kappa架构

考虑到由爱立信公司提供技术支持的运营商通常拥有庞大的数据规模（每天处理 10TB~100TB 的数据，每秒处理 10 万 ~100 万个事件），该公司的一支团队决定实现所谓的 Kappa 架构[3]。2014 年，Kafka 的创始人之一 Jay Kreps 为 O’Reilly Radar 撰写了一篇批评 Lambda 架构的文章[4]，并在其中开玩笑式地创造了“Kappa 架构”这个词。其实，Kappa 架构正是第 2 章所讨论的流处理架构。其中，数据流是设计核心；数据源不可变更；架构采用像 Flink 这样的单一流分析框架处理新鲜数据，并通过流重播处理历史数据。

爱立信公司需要实时分析云基础设施的系统性能指标和日志，从而持续地监视系统行为，以确定是一切正常，还是有“新奇点”出现。“新奇点”既可能是异常行为，也可能是系统状态变更，例如加入了新虚拟机。爱立信团队使用的方法是将一个贝叶斯在线学习模型应用于包含电信云监控系统多个指标的数据流（遥测信息和日志事件）。爱立信公司的研究人员 Nicolas Seyvet 和 Ignacio Mulas Viela 说道：

> 该架构在不断地适应（学习）新系统常态的同时，能够快速且准确地发现异常。这使它成为理想工具，并能够极大地降低因大型计算设施运行而产生的维护成本。

图 4-8 展示了爱立信团队构建的数据管道。

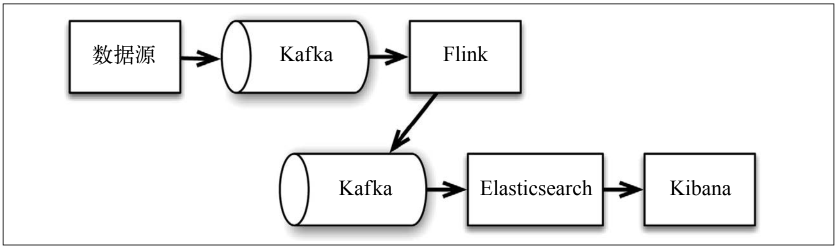



**图 4-8：爱立信团队采用的基于 Flink 的流处理架构**

---

注 3：本节内容基于 Nicolas Seyvet 和 Ignacio Mulas Viela 在 Strata + Hadoop World London 2016 研讨会上所做的演讲。Ignacio Mulas Viela 在 2015 年的 Flink Forward 研讨会上也做过相关的演讲。

注 4：https://www.oreilly.com/ideas/questioning-the-lambda-architecture

推送给 Kafka 的原始数据是来自云基础设施中的所有实体机和虚拟机的遥测信息和日志事件。它们经过不同的 Flink 作业消费之后，被写回 Kafka 主题里，然后再从 Kafka 主题里被推送给搜索引擎 Elasticsearch 和可视化系统 Kibana。这种架构让每个 Flink 作业所执行的任务有清晰的定义，一个作业的输出可以成为另一个作业的输入。图 4-9 展示了异常检测管道，每个中间流都是 Kafka 主题（以分配给它的数据命名），每个长方形代表一个 Flink 作业。

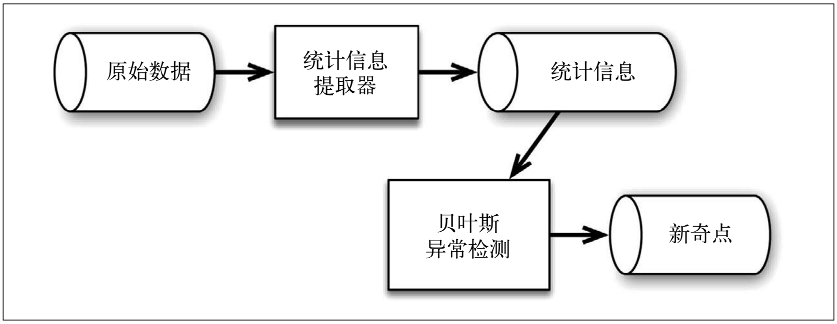



图 4-9：爱立信公司的异常检测管道采用 Flink 实现统计信息提取和异常检测

在本案例中，为什么 Flink 对事件时间的支持很重要呢？有两个原因。

(1) *有助于准确地识别异常*。时间对识别异常很重要。当许多日志事件在同一时间出现时，通常说明可能有错误发生。为了将这些事件正确地分组和归类，考虑它们的真实时间（而不是处理时间）很重要。
(2) *有助于采用流处理架构*。在流处理架构中，所有的计算都由流处理器完成。升级应用程序的做法是将它们在流处理器中再次执行。用同一种计算运行两次同样的数据，必须得到同样的结果，这只有依靠事件时间操作才能实现。

# 第 5 章

# 有状态的计算

流式计算分为无状态和有状态两种情况。无状态的计算观察每个独立事件，并根据最后一个事件输出结果。例如，流处理应用程序从传感器接收温度读数，并在温度超过 90 度时发出警告。有状态的计算则会基于多个事件输出结果。以下是一些例子。

- 第 4 章讨论的所有类型的窗口。例如，计算过去一小时的平均温度，就是有状态的计算。
- 所有用于复杂事件处理的状态机。例如，若在一分钟内收到两个相差 20 度以上的温度读数，则发出警告，这是有状态的计算。
- 流与流之间的所有关联操作，以及流与静态表或动态表之间的关联操作，都是有状态的计算。

图 5-1 展示了无状态流处理和有状态流处理的主要区别。无状态流处理分别接收每条记录（图中的黑条），然后根据最新输入的记录生成输出记录（白条）。有状态流处理会维护状态（根据每条输入记录进行更新），并基于最新输入的记录和当前的状态值生成输出记录（灰条）。

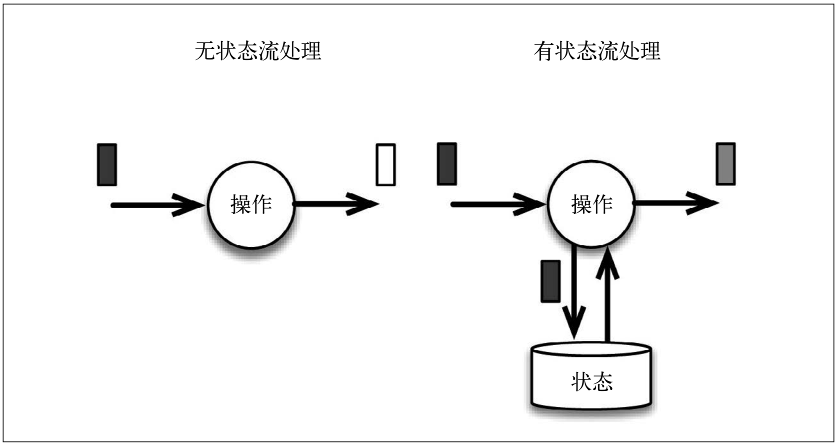



图 5-1：无状态流处理与有状态流处理的区别。输入记录由黑条表示。无状态流处理每次只转换一条输入记录，并且仅根据最新的输入记录输出结果（白条）。有状态流处理维护所有已处理记录的状态值，并根据每条新输入的记录更新状态，因此输出记录（灰条）反映的是综合考虑多个事件之后的结果

尽管无状态的计算很重要，但是流处理对有状态的计算更感兴趣。事实上，正确地实现有状态的计算比实现无状态的计算难得多。旧的流处理系统并不支持有状态的计算，而新一代的流处理系统则将状态及其正确性视为重中之重。

## 5.1　一致性

当在分布式系统中引入状态时，自然也引入了一致性问题。一致性实际上是“正确性级别”的另一种说法，即在成功处理故障并恢复之后得到的结果，与没有发生任何故障时得到的结果相比，前者有多正确？举例来说，假设要对最近一小时登录的用户计数。在系统经历故障之后，计数结果是多少？在流处理中，一致性分为 3 个级别。

- at-most-once：这其实是没有正确性保障的委婉说法——故障发生之后，计数结果可能丢失。
- at-least-once：这表示计数结果可能大于正确值，但绝不会小于正确值。也就是说，计数程序在发生故障后可能多算，但是绝不会少算。

- exactly-once：这指的是系统保证在发生故障后得到的计数结果与正确值一致。

曾经，at-least-once 非常流行。第一代流处理器（如 Storm 和 Samza）刚问世时只保证 at-least-once，原因有二。

(1) 保证 exactly-once 的系统实现起来更复杂。这在基础架构层（决定什么代表正确，以及 exactly-once 的范围是什么）和实现层都很有挑战性。
(2) 流处理系统的早期用户愿意接受框架的局限性，并在应用层想办法弥补（例如使应用程序具有幂等性，或者用批量计算层再做一遍计算）。

最先保证 exactly-once 的系统（Storm Trident 和 Spark Streaming）在性能和表现力这两个方面付出了很大的代价。为了保证 exactly-once，这些系统无法单独地对每条记录运用应用逻辑，而是同时处理多条（一批）记录，保证对每一批的处理要么全部成功，要么全部失败。这就导致在得到结果前，必须等待一批记录处理结束。因此，用户经常不得不使用两个流处理框架（一个用来保证 exactly-once，另一个用来对每个元素做低延迟处理），结果使基础设施更加复杂。曾经，用户不得不在保证 exactly-once 与获得低延迟和效率之间权衡利弊。Flink 避免了这种权衡。

> Flink 的一个重大价值在于，它既保证了 exactly-once，也具有低延迟和高吞吐的处理能力。

从根本上说，Flink 通过使自身满足所有需求来避免权衡，它是业界的一次意义重大的技术飞跃。尽管这在外行看来很神奇，但是一旦了解，就会恍然大悟。

## 5.2 检查点：保证exactly-once

Flink 如何保证 exactly-once 呢？它使用一种被称为“检查点”的特性，在出现故障时将系统重置回正确状态。下面通过简单的类比来解释检查点的作用。

假设你和两位朋友正在数项链上有多少颗珠子，如图 5-2 所示。你捏住珠

子，边数边拨，每拨过一颗珠子就给总数加一。你的朋友也这样数他们手中的珠子。当你分神忘记数到哪里时，怎么办呢？如果项链上有很多珠子，你显然不想从头再数一遍，尤其是当三人的速度不一样却又试图合作的时候，更是如此（比如想记录前一分钟三人一共数了多少颗珠子，回想一下第 4 章中的一分钟滚动窗口）。

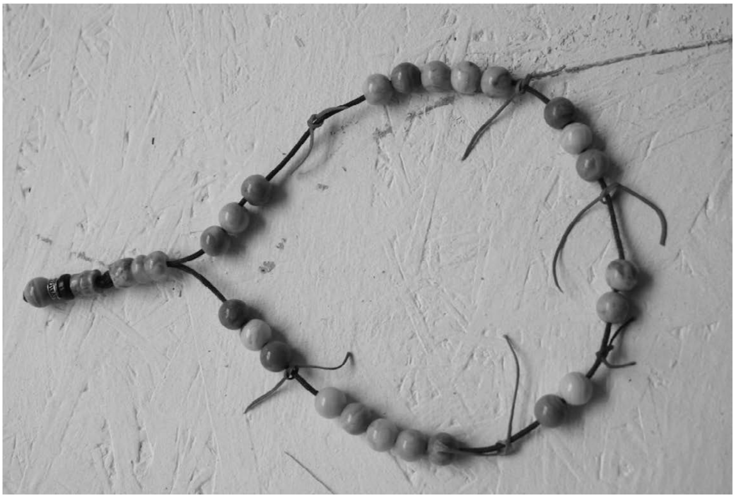


图 5-2：数环状项链上的珠子看上去毫无意义（甚至有些徒劳无功，因为可以永不停歇地计数），但是它可以用来很好地类比处理永不结束的事件流。在某些文化中，人们仍旧将数珠子视作消磨时间的好方法

于是，你想了一个更好的办法：在项链上每隔一段就松松地系上一根有色皮筋，将珠子分隔开；当珠子被拨动的时候，皮筋也可以被拨动；然后，你安排一个助手，让他在你和朋友拨到皮筋时记录总数。用这种方法，当有人数错时，就不必从头开始数。相反，你向其他人发出错误警示，然后你们都从上一根皮筋处开始重数，助手则会告诉每个人重数时的起始数值，例如在粉色皮筋处的数值是多少。

Flink 检查点的作用就类似于皮筋标记。数珠子这个类比的关键点是：对于指定的皮筋而言，珠子的相对位置是确定的；这让皮筋成为重新计数的参考点。总状态（珠子的总数）在每颗珠子被拨动之后更新一次，助手则会

保存与每根皮筋对应的检查点状态，如当遇到粉色皮筋时一共数了多少珠子，当遇到橙色皮筋时又是多少。当问题出现时，这种方法使得重新计数变得简单。

> 检查点是 Flink 最有价值的创新之一，因为它使 Flink 可以保证 exactly-once，并且不需要牺牲性能。

Flink 检查点的核心作用是确保状态正确，即使遇到程序中断，也要正确。记住这一基本点之后，我们用一个例子来看检查点是如何运行的。Flink 为用户提供了用来定义状态的工具。例如，以下这个 Scala 程序按照输入记录的第一个字段（一个字符串）进行分组并维护第二个字段的计数状态。

```
val stream: DataStream[(String, Int)] = ...

val counts: DataStream[(String, Int)] = stream
  .keyBy(record => record._1)
  .mapWithState((in: (String, Int), count: Option[Int]) =>
    count match {
      case Some(c) => ( (in._1, c + in._2), Some(c + in._2) )
      case None => ( (in._1, in._2), Some(in._2) )
    })
```

该程序有两个算子：`keyBy` 算子用来将记录按照第一个元素（一个字符串）进行分组，根据该 key 将数据进行重新分区，然后将记录再发送给下一个算子：有状态的 `map` 算子（`mapWithState`）。`map` 算子在接收到每个元素后，将输入记录的第二个字段的数据加到现有总数中，再将更新过的元素发射出去。图 5-3 表示程序的初始状态：输入流中的 6 条记录被检查点屏障（checkpoint barrier）隔开，所有的 `map` 算子状态均为 0（计数还未开始）。所有 key 为 a 的记录将被顶层的 `map` 算子处理，所有 key 为 b 的记录将被中间层的 `map` 算子处理，所有 key 为 c 的记录则将被底层的 `map` 算子处理。

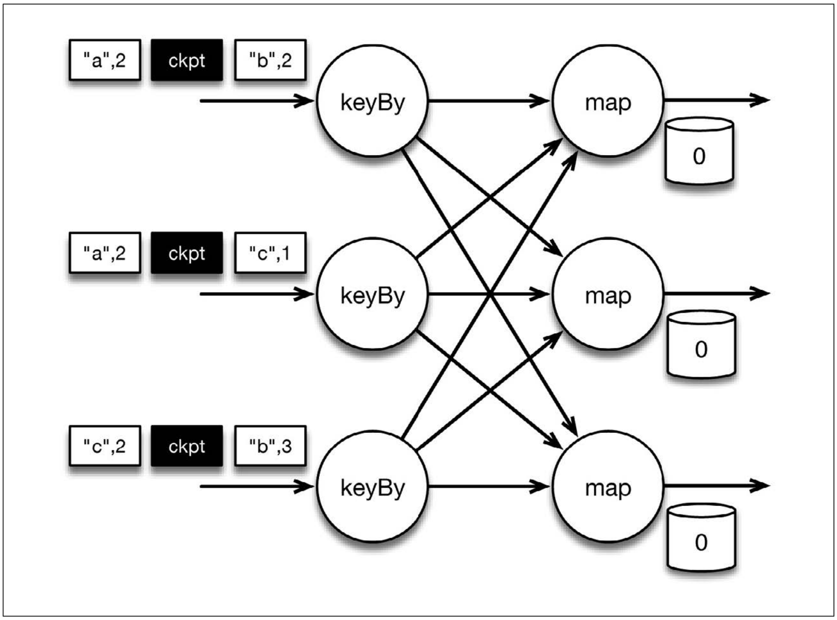
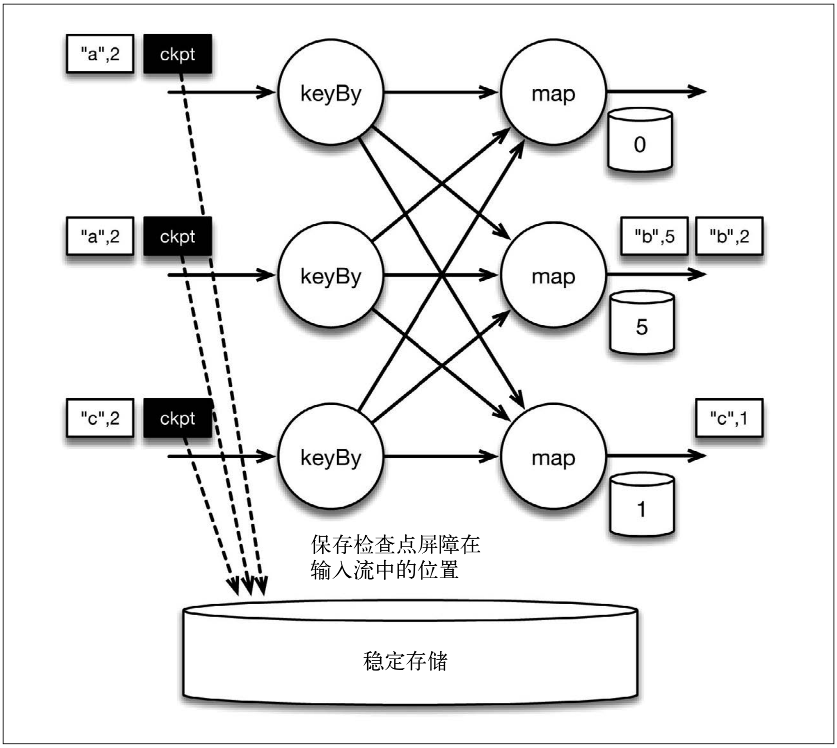



图 5-3：程序的初始状态。注意，a、b、c 三组的初始计数状态都是 0，即三个圆柱上的值。ckpt 表示检查点屏障。每条记录在处理顺序上严格地遵守在检查点之前或之后的规定，例如 ["b",2] 在检查点之前被处理，["a",2] 则在检查点之后被处理

当该程序处理输入流中的 6 条记录时，涉及的操作遍布 3 个并行实例（节点、CPU 内核等）。那么，检查点该如何保证 exactly-once 呢？

检查点屏障和普通记录类似。它们由算子处理，但并不参与计算，而是会触发与检查点相关的行为。当读取输入流的数据源（在本例中与 keyBy 算子内联）遇到检查点屏障时，它将其在输入流中的位置保存到稳定存储中。如果输入流来自消息传输系统（Kafka 或 MapR Streams），这个位置就是偏移量。Flink 的存储机制是插件化的，稳定存储可以是分布式文件系统，如 HDFS、S3 或 MapR-FS。图 5-4 展示了这个过程。




邮
电

图 5-4：当 Flink 数据源（在本例中与 keyBy 算子内联）遇到检查点屏障时，它会将其在输入流中的位置保存到稳定存储中。这让 Flink 可以根据该位置重启输入

检查点屏障像普通记录一样在算子之间流动。当 map 算子处理完前 3 条记录并收到检查点屏障时，它们会将状态以异步的方式写入稳定存储，如图 5-5 所示。

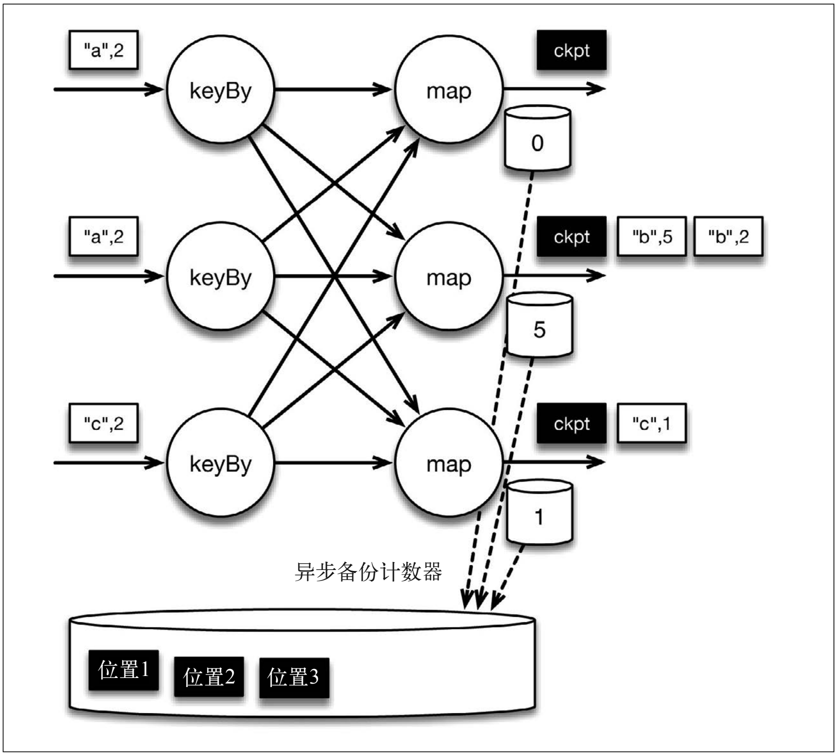



图 5-5：位于检查点之前的所有记录（["b",2]、["b",3] 和 ["c",1]）被 map 算子处理之后的情况。此时，稳定存储已经备份了检查点屏障在输入流中的位置（备份操作发生在检查点屏障被输入算子处理的时候）。map 算子接着开始处理检查点屏障，并触发将状态异步备份到稳定存储中这个动作

当 map 算子的状态备份和检查点屏障的位置备份被确认之后，该检查点操作就可以被标记为完成，如图 5-6 所示。我们在无须停止或者阻断计算的条件下，在一个逻辑时间点（对应检查点屏障在输入流中的位置）为计算状态拍了快照。通过确保备份的状态和位置指向同一个逻辑时间点，后文将解释如何基于备份恢复计算，从而保证 exactly-once。值得注意的是，当没有出现故障时，Flink 检查点的开销极小，检查点操作的速度由稳定存储的可用带宽决定。回顾数珠子的例子：除了因为数错而需要用到皮筋之外，皮筋会被很快地拨过。（Flink 的开发团队正在研究如何只保存状态的变化，而不保存状态的值，这样做可以让开销变得更小。）

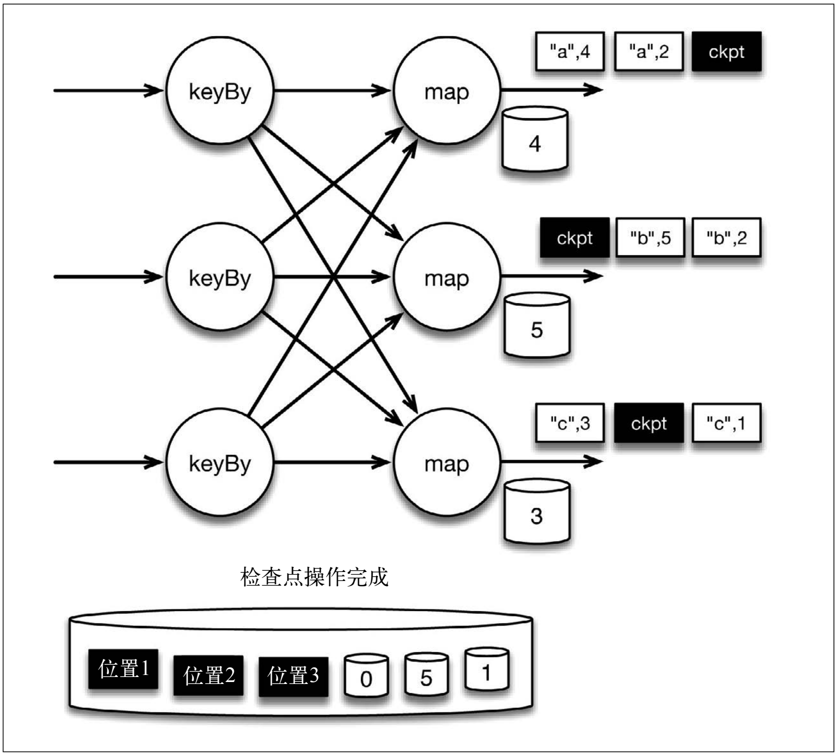



图 5-6：检查点操作完成，状态和位置均已备份到稳定存储中。输入流中的所有记录都已处理完成。值得注意的是，备份的状态值与实际的状态值是不同的。备份反映的是检查点的状态

如果检查点操作失败，Flink 会丢弃该检查点并继续正常执行，因为之后的某一个检查点可能会成功。虽然恢复时间可能更长，但是对于状态的保证依旧很有力。只有在一系列连续的检查点操作失败之后，Flink 才会抛出错误，因为这通常预示着发生了严重且持久的错误。

现在来看看图 5-7 所示的情况：检查点操作已经完成，但故障紧随其后。

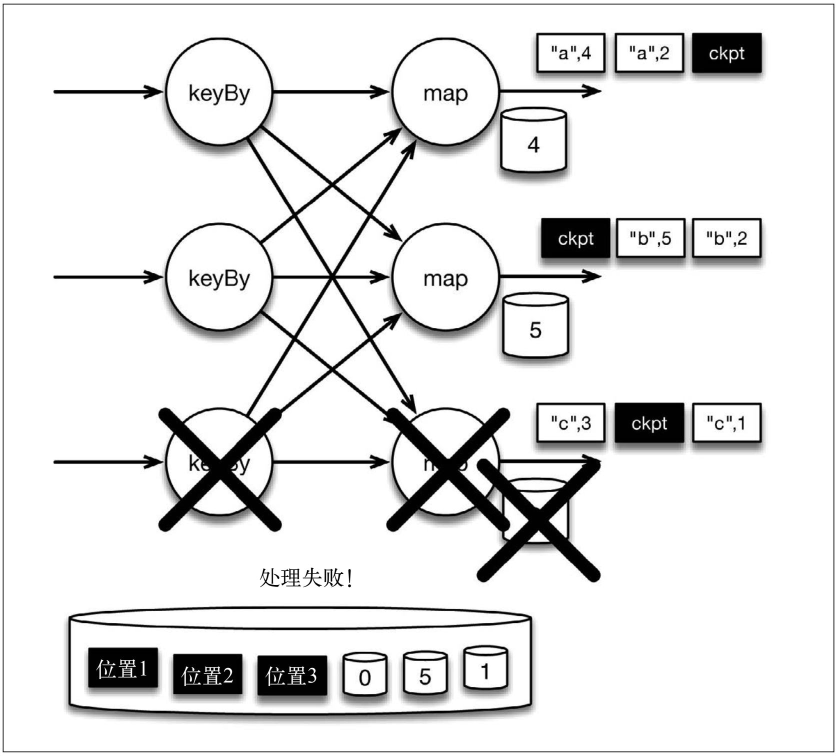



图 5-7：故障紧跟检查点，导致最底部的实例丢失

在这种情况下，Flink 会重新拓扑（可能会获取新的执行资源），将输入流倒回到上一个检查点，然后恢复状态值并从该处开始继续计算。在本例中，["a",2]、["a",2] 和 ["c",2] 这几条记录将被重播。

图 5-8 展示了这一重新处理过程。从上一个检查点开始重新计算，可以保证在剩下的记录被处理之后，得到的 `map` 算子的状态值与没有发生故障时的状态值一致。值得注意的是，输出流会含有重复的数据。具体来说，["a",2]、["a",4] 和 ["c",3] 会出现两次。如果 Flink 将输出流写入特殊的输出系统（比如文件系统或者数据库），那么就可以避免这个问题，本章稍后将进一步讨论。

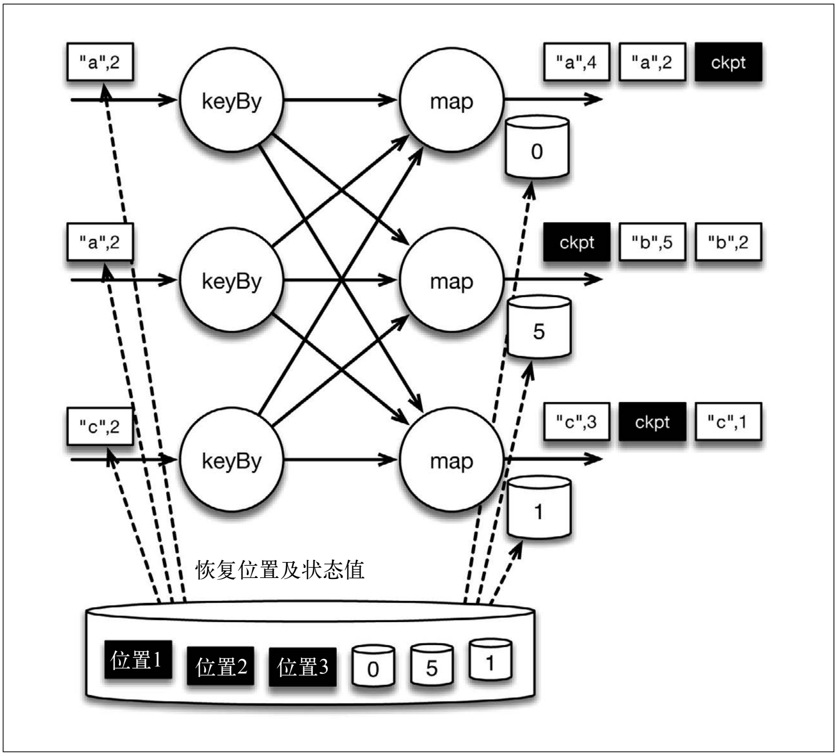



图 5-8：Flink 将输入流倒回到上一个检查点屏障的位置，同时恢复 map 算子的状态值。然后，Flink 从此处开始重新处理。这样做保证了在记录被处理之后，map 算子的状态值与没有发生故障时的一致

Flink 检查点算法的正式名称是*异步屏障快照*（asynchronous barrier snapshotting）。该算法大致基于 Chandy-Lamport 分布式快照算法。

## 5.3 保存点：状态版本控制

检查点由 Flink 自动生成，用来在故障发生时重新处理记录，从而修正状态。Flink 用户还可以通过另一个特性有意识地管理状态版本，这个特性叫作*保存点*（savepoint）。

保存点与检查点的工作方式完全相同，只不过它由用户通过 Flink 命令行工具或者 Web 控制台手动触发，而不由 Flink 自动触发。和检查点一样，保存点也被保存在稳定存储中。用户可以从保存点重启作业，而不用从头开始。保存点可以被视为作业在某一个特定时间点的快照（该时间点即为保存点被触发的时间点）。

对保存点的另一种理解是，它在明确的时间点保存应用程序状态的版本。这和用版本控制系统保存应用程序的版本很相似。最简单的例子是在不修改应用程序代码的情况下，每隔固定的时间拍快照，即照原样保存应用程序状态的版本，如图 5-9 所示。

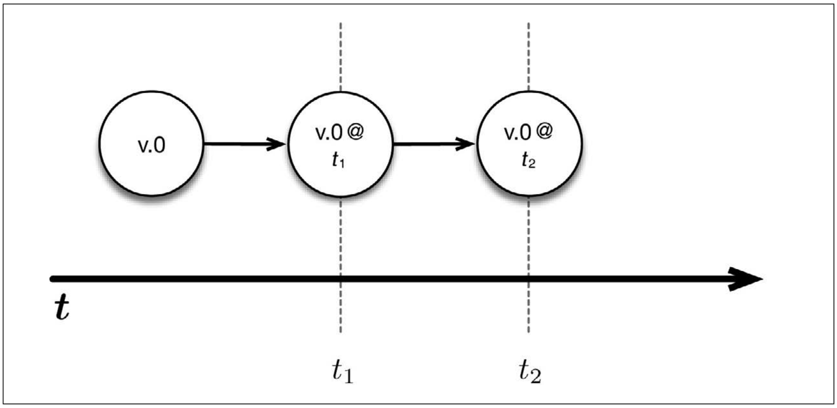



图 5-9：手动触发的保存点（以圆圈表示）在不同时间捕获正在运行的 Flink 应用程序的状态

在图中，v.0 是某应用程序的一个正在运行的版本。我们分别在 $t_1$ 时刻和 $t_2$ 时刻触发了保存点。因此，可以在任何时候返回到这两个时间点，并且重启程序。更重要的是，可以从保存点启动被修改过的程序版本。举例来说，可以修改应用程序的代码（假设称新版本为 v.1），然后从 $t_1$ 时刻开始运行改动过的代码。这样一来，v.0 和 v.1 这两个版本同时运行，并在之后的时间里获取各自的保存点，如图 5-10 所示。

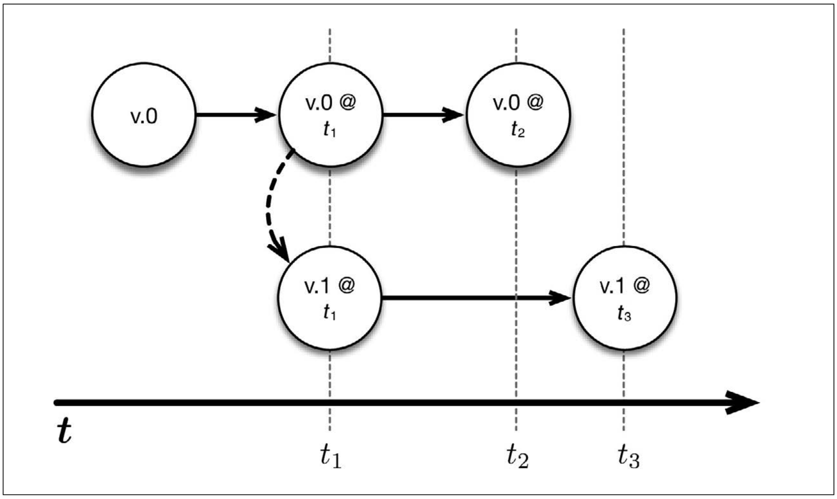



图 5-10：使用保存点更新 Flink 应用程序的版本。新版本可以从旧版本生成的一个保存点处开始执行

保存点可用于应对流处理作业在生产环境中遇到的许多挑战。

(1) 应用程序代码升级：假设你在已经处于运行状态的应用程序中发现了一个 bug，并且希望之后的事件都可以用修复后的新版本来处理。通过触发保存点并从该保存点处运行新版本，下游的应用程序并不会察觉到不同（当然，被更新的部分除外）。

(2) Flink 版本更新：Flink 自身的更新也变得简单，因为可以针对正在运行的任务触发保存点，并从保存点处用新版本的 Flink 重启任务。

(3) 维护和迁移：使用保存点，可以轻松地“暂停和恢复”应用程序。这对于集群维护以及向新集群迁移的作业来说尤其有用。此外，它还有利于开发、测试和调试，因为不需要重播整个事件流。

(4) 假设模拟与恢复：在可控的点上运行其他的应用逻辑，以模拟假设的场景，这样做在很多时候非常有用。

(5) A/B 测试：从同一个保存点开始，并行地运行应用程序的两个版本，有助于进行 A/B 测试。

上述所有挑战都真实存在。Flink 内部的检查点机制以保存点的形式呈现给用户，用来应对上述挑战。这反映了 Flink 检查点本质上是一个可持续升

级状态版本的可编程机制，这一点很像具有多版本并发控制的数据库系统。下一节在讨论如何提供端到端的一致性时，还会提到检查点机制的这一基本特征。

## 5.4 端到端的一致性和作为数据库的流处理器

我们已经通过简单的计数例子了解了 Flink 如何保证状态的一致性（即保证 exactly-once）。接下来看看端到端的情况，因为在生产环境中可能会部署这种应用程序（如图 5-11 所示）。

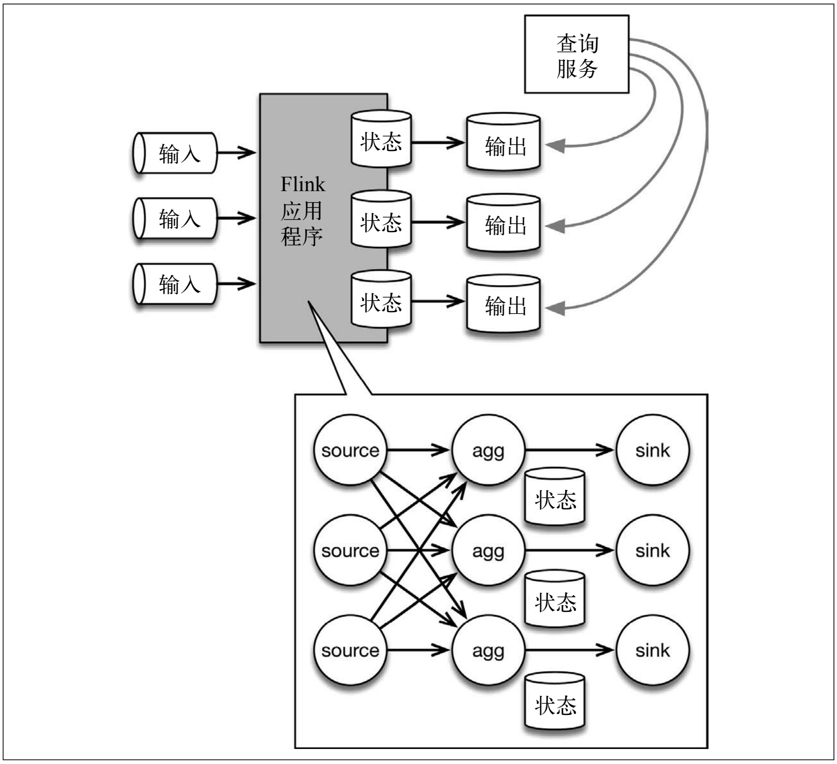



图 5-11：在该应用程序架构中，有状态的 Flink 应用程序消费来自消息队列的数据，然后将数据写入输出系统，以供查询。底部的详情图展示了 Flink 应用程序的内部情况

输入数据来自一个分区存储系统（如 Kafka 或者 MapR Streams 这样的消息队列）。图 5-11 底部的详情图展示了 Flink 拓扑，其中包含 3 个算子。source 读取输入数据，根据 key 分区，并将数据路由到有状态的算子实例（这既可以是 5.2 节提到的 `map` 算子，也可以是窗口聚合算子）。有状态的算子将状态内容（比如前例中的计数结果）或者一些衍生结果写入 sink，再由 sink 将结果传送到输出存储系统中（例如文件系统或数据库）。接着，查询服务（比如数据库查询 API）就可以允许用户对状态进行查询（最简单的例子就是查询计数结果），因为状态已经被写入输出存储系统了。

> 需要记住的是，在本例中，输出反映的是截至最近一次写入状态之时，Flink 应用程序中的状态内容。

在将状态内容传送到输出存储系统的过程中，如何保证 exactly-once 呢？这叫作端到端的一致性。本质上有两种实现方法，用哪一种方法则取决于输出存储系统的类型，以及应用程序的需求。

(1) 第一种方法是在 sink 环节缓冲所有输出，并在 sink 收到检查点记录时，将输出“原子提交”到存储系统。这种方法保证输出存储系统中只存在有一致性保障的结果，并且不会出现重复的数据。从本质上说，输出存储系统会参与 Flink 的检查点操作。要做到这一点，输出存储系统需要具备“原子提交”的能力。

(2) 第二种方法是急切地将数据写入输出存储系统，同时牢记这些数据可能是“脏”的，而且需要在发生故障时重新处理。如果发生故障，就需要将输出、输入和 Flink 作业全部回滚，从而将“脏”数据覆盖，并将已经写入输出的“脏”数据删除。注意，在很多情况下，其实并没有发生删除操作。例如，如果新记录只是覆盖旧纪录（而不是添加到输出中），那么“脏”数据只在检查点之间短暂存在，并且最终会被修正过的新数据覆盖。

值得注意的是，这两种方法恰好对应关系数据库系统中的两种为人所熟知的事务隔离级别：*已提交读*（read committed）和*未提交读*（read uncommitted）。已提交读保证所有读取（查询输出）都只读取已提交的数据，而不会读取中间、传输中或“脏”的数据。之后的读取可能会返回不同的结果，因为数据可能已被改变。未提交读则允许读取“脏”数据；换句话说，查询总

是看到被处理过的最新版本的数据。

某些应用程序可以接受弱一点的语义，所以 Flink 提供了支持多重语义的多种内置输出算子，如支持未提交读语义的分布式文件输出算子。用户可以根据输出存储系统的能力和应用程序的需求选择合适的语义。

根据输出存储系统的类型，Flink 及与之对应的连接器可以一起保证端到端的一致性，并且支持多种隔离级别。

现在回过头看看图 5-11 中的应用程序架构。之所以本例需要有输出存储系统，是因为外部无法访问 Flink 的内部状态，所以输出存储系统成了查询目标。但是，如果可以直接查询状态，则在某些情况下根本就不需要输出存储系统，因为状态本身就已经包含了查询所需的信息。这种情况在许多应用程序中真实存在，直接查询状态可以大大地简化架构，同时大幅提升性能，如图 5-12 所示。

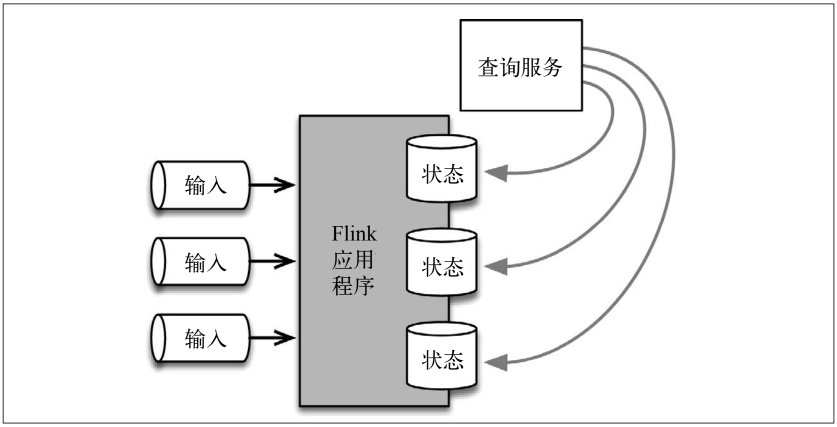



图 5-12：利用 Flink 的可查询状态特性可以简化应用程序架构。对于状态本身就是所需信息的查询来说，直接查询状态可以提升性能

Flink 社区正致力于完善可查询状态特性。Flink 提供一个查询 API，通过该 API 可以对 Flink 发出查询请求，然后得到当前的状态值。从某种意义上说，在有限的情景下，Flink 可以替代数据库，并同时提供写路径（输入流不断更新状态）和读路径（可查询状态）。尽管这对于许多应用程序都行得通，但可查询状态受到的限制还是比通用数据库大得多。

# 5.5 Flink的性能

Flink 的性能测试基于 Yahoo! Streaming Benchmark 及其一系列变体进行。

## 5.5.1 Yahoo! Streaming Benchmark

2015 年 12 月，Yahoo! 的 Storm 团队发表了一篇博客文章 [1]，并在其中展示了 Storm、Flink 和 Spark Streaming 的性能测试结果。该测试对于业界而言极具价值，因为它是流处理领域的第一个基于真实应用程序的基准测试。

该应用程序从 Kafka 消费广告曝光消息，从 Redis 查找每个广告对应的广告宣传活动，并按照广告宣传活动分组，以 10 秒为窗口计算广告浏览量。10 秒窗口的最终结果被存储在 Redis 中，这些窗口的状态也按照每秒记录一次的频率被写入 Redis，以方便用户对它们进行实时查询。在最初的性能测评中，因为 Storm 是无状态流处理器（即它不能定义和维护状态），所以 Flink 作业也按照无状态模式编写。所有状态都被存储在 Redis 中，如图 5-13 所示。

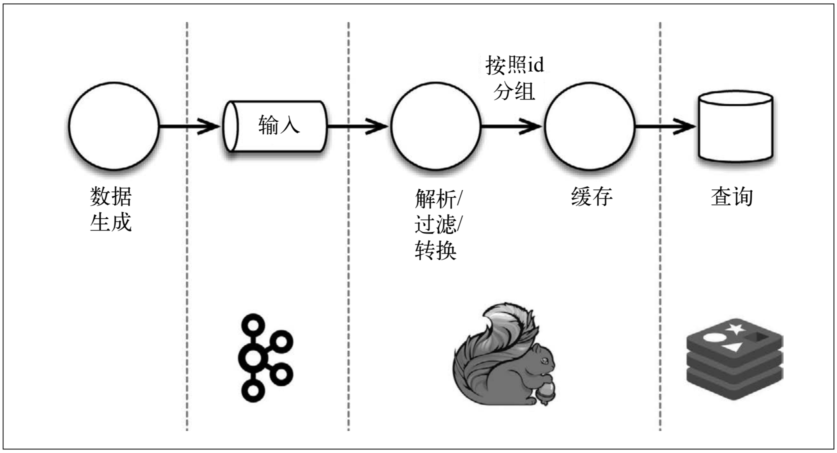



图 5-13：Yahoo! Streaming Benchmark 所采用的作业。被测试的流处理器有 Storm、Flink 和 Spark Streaming（本图中只有 Flink 的 logo）

注 1：https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

图 5-14 展示了测试结果。

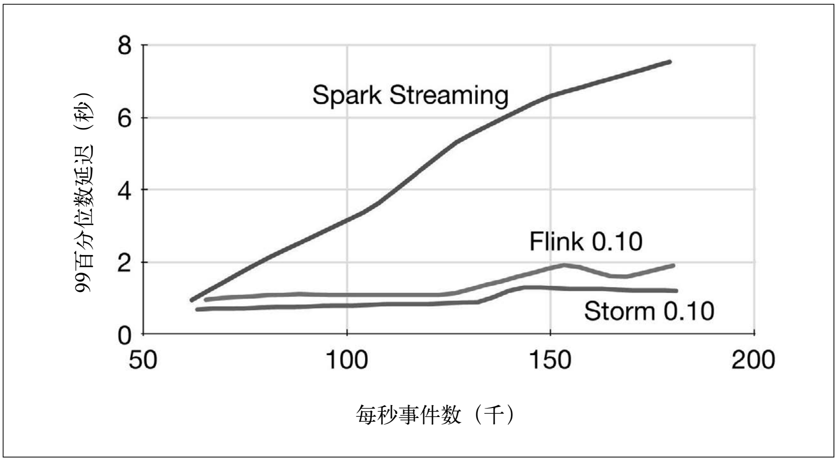



图 5-14：Yahoo! Streaming Benchmark 的结果。横轴表示每秒的事件吞吐量，以千为单位。纵轴表示端到端的 99 百分位数延迟（即 99% 的事件在延迟时间段内到达），以秒为单位。详见博客文章 *Benchmarking Streaming Computation Engines at Yahoo!* [2]

如图 5-14 所示，在性能测评中，Spark Streaming 遇到了吞吐量和延迟性难两全的问题。随着批处理作业规模的增加，延迟升高。如果为了降低延迟而缩减规模，吞吐量就会减少。Storm 和 Flink 则可以在吞吐量增加时维持低延迟。

为了进一步测试 Flink 的性能，测试人员设置了一系列不同的场景，并逐步测试。

## 5.5.2 变化1：使用Flink状态

最初的性能测评专注于在相对较低的吞吐量下，测量端到端的延迟，即使在极限状态下，也不关注容错性。此外，应用程序中的 key 基数非常小（100），这使得测试结果无法反映用户量大的情况，或者 key 空间随着时间

注 2：https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

增长的情况（如 tweet）。2016 年 2 月，data Artisans 的博客发表了一篇文章[3]，对 Yahoo! Streaming Benchmark 进行了拓展，并专注于解决上述问题。由于最初的测试结果显示 Spark Streaming 的性能欠佳，因此这次的测试对象只有 Storm 和 Flink，它们在最初的测试中有着类似的表现。

第 1 个变化是利用 Flink 提供的状态容错特性重新实现应用程序，如图 5-15 所示。这使得应用程序能保证 exactly-once。

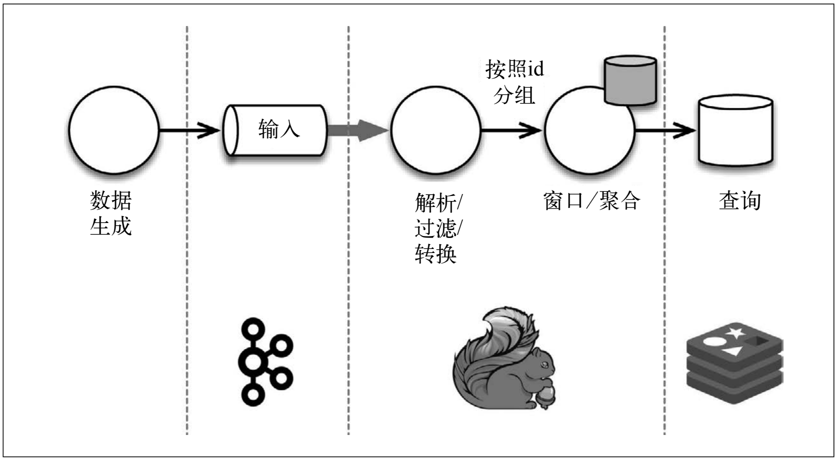



图 5-15：重新实现的应用程序利用了 Flink 内置的状态机制，并且可以保持每秒 300 万事件的吞吐量，同时保证 exactly-once。此时，应用程序的瓶颈在于 Flink 集群与 Kafka 集群的连接（图中以粗箭头表示）

## 5.5.3 变化2：改进数据生成器并增加吞吐量

第 2 个变化是通过用每秒可以生成数百万事件的数据生成器来增加输入流的数据量。结果如图 5-16 所示。

注 3：https://data-artisans.com/blog/extending-the-yahoo-streaming-benchmark

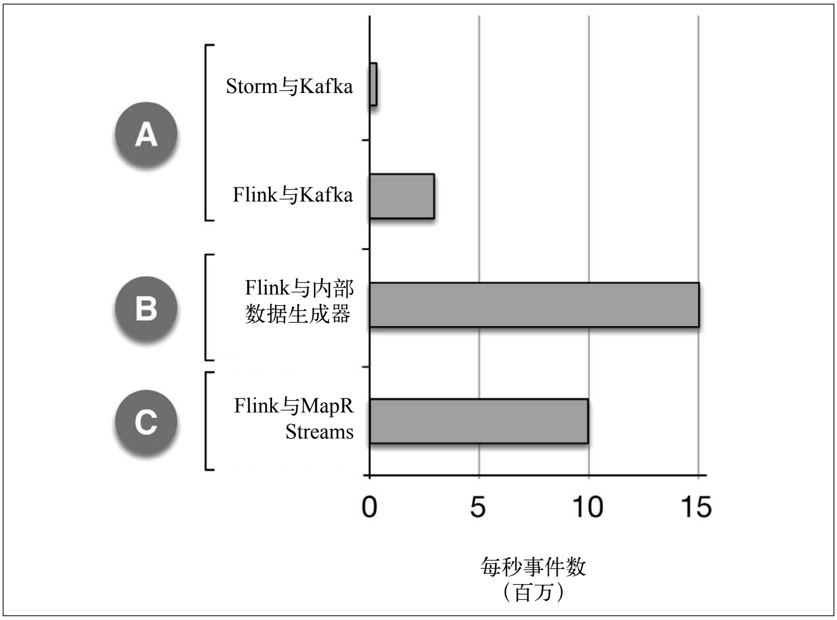



图 5-16：使用高吞吐数据生成器的结果：（A）当 Storm 与 Kafka 一起使用时，应用程序可以保持每秒 40 万事件的处理速度，并且瓶颈在于 CPU；当 Flink 与 Kafka 一起使用时，应用程序可以保持每秒 300 万事件的处理速度，并且瓶颈在于网络；（B）当消除网络瓶颈时，Flink 应用程序可以保持每秒 1500 万事件的处理速度；（C）在额外的测试中，消息队列由 MapR Streams 提供，并且采用 10 个高性能网络节点（硬件与前两种情况中的不同）；Flink 应用程序可以保持每秒 1000 万事件的处理速度

Storm 能够承受每秒 40 万事件，但受限于 CPU；Flink 则可以达到每秒 300 万事件（7.5 倍），但受限于 Kafka 集群和 Flink 集群之间的网络。

## 5.5.4 变化3：消除网络瓶颈

为了看看在没有网络瓶颈问题时 Flink 的性能如何，我们将数据生成器移到 Flink 应用程序的内部。图 5-17 展示了这个流程。在这样的条件下，Flink 可以保持每秒 1500 万事件的处理速度（这是 Storm 的 37.5 倍），如图 5-16 所示。将数据生成器整合到 Flink 应用程序中，可以测试性能极限，但这种做法并不现实，因为现实世界中的数据必须从应用程序的外部流入。

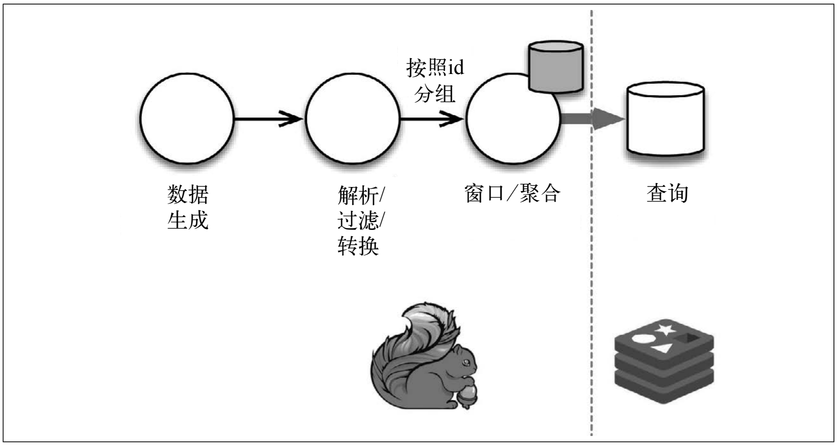



图 5-17：通过将数据生成器整合到 Flink 应用程序中，可以消除网络瓶颈，并且使系统支撑每秒 1500 万事件的吞吐量。在增加 key 基数之后，瓶颈又转移到每秒对 Redis 的写入上。该测试并不符合生产环境的配置，它的目的是测试 Flink 的极限

值得注意的是，这绝对不是 Kafka 的极限（Kafka 可以支撑比这更大的吞吐量），而仅仅是测试所用的硬件环境的极限——Kafka 集群和 Flink 集群之间的网络连接太慢。

## 5.5.5　变化4：使用MapR Streams

另一种避免网络瓶颈并测试 Flink 性能的方法是使用 MapR Streams。在另一个测试中，同样的 Flink 应用程序通过 MapR Streams 接收数据。

使用 MapR Streams 之后，流处理被整合进整个平台，从而使得 Flink 可以与数据生成任务和数据传输任务一起运行，这样就避免了连接 Kafka 集群和 Flink 集群时遇到的大部分问题。在这种高性能配置和更快的网络硬件环境下，Flink 能够支撑每秒 1000 万事件的处理速度。

## 5.5.6　变化5：增加key基数

最后一个变化是增加 key 基数（广告宣传活动的数量）。在最初的测试中，key 基数只有 100。这些 key 每秒都会被写入 Redis，以供查询。当 key 基数增加到 100 万时，系统的整体吞吐量减少到每秒 28 万事件，因为向 Redis

写入成了系统瓶颈。使用 Flink 可查询状态的一个早期原型（如图 5-18 所示），可以消除这种瓶颈，使系统的处理速度恢复到每秒 1500 万事件，并且有 100 万个 key 可供查询，如图 5-19 所示。

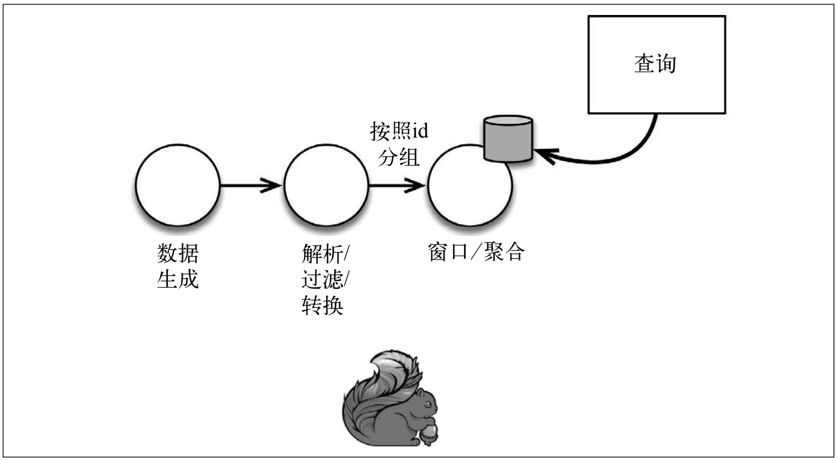



图 5-18：为 key 基数较大的场景消除系统瓶颈

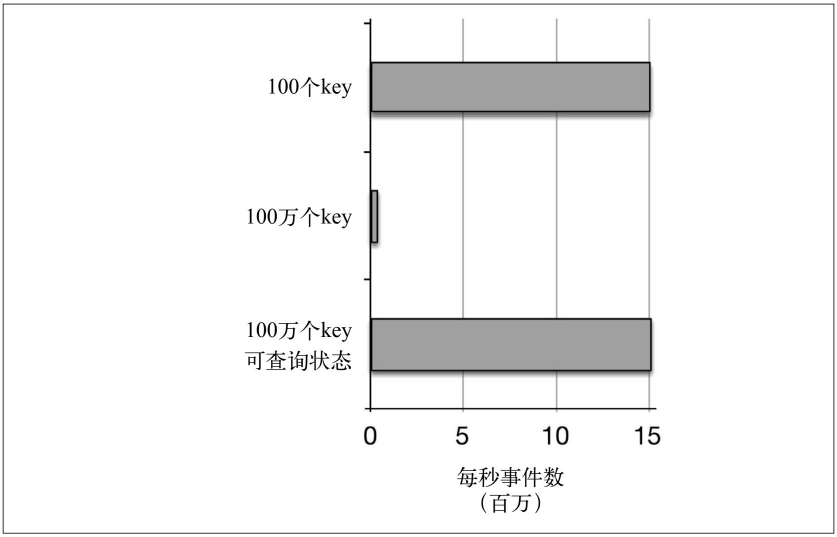



图 5-19：通过将查询功能移入 Flink 可查询状态的一个原型，系统甚至可以在 key 基数非常大的情况下仍然维持每秒 1500 万事件的处理速度

本例说明了什么呢？通过避免流处理瓶颈，同时利用 Flink 的有状态流处理能力，可以使吞吐量达到 Storm 的 30 倍左右，同时还能保证 exactly-once 和高可用性。大致来说，这意味着与 Storm 相比，Flink 的硬件成本或云计算成本仅为前者的 1/30，同样的硬件能处理的数据量则是前者的 30 倍。

# 5.6　结论

在本章中，我们看到有状态的流处理如何改变了游戏规则。通过 Flink 的检查点机制，能够同时实现容错、高吞吐和低延迟。这完全避免了人们曾经认为无法避免的权衡，并体现了 Flink 最重要的一个优势。

Flink 的另一个优势是，它用同一种技术实现流处理和批处理，因此完全不必再建一个批处理层。第 6 章将简述 Flink 如何实现批处理。

# 第 6 章

# 批处理：一种特殊的流处理

到目前为止，本书讨论的都是*无限流处理*，即从某个时间点开始，持续不停地处理数据，如图 6-1 所示。

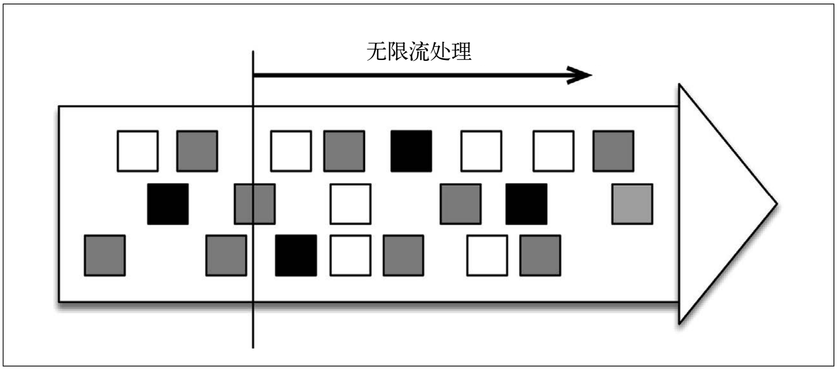



图 6-1：无限流处理：输入数据没有尽头；数据处理从当前或者过去的某一个时间点开始，持续不停地进行

另一种处理形式叫作*有限流处理*，即从某一个时间点开始处理数据，然后在另一个时间点结束，如图 6-2 所示。输入数据可能本身是有限的（即输入数据集并不会随着时间增长），也可能出于分析的目的被人为地设定为有限集（即只分析某一个时间段内的事件）。

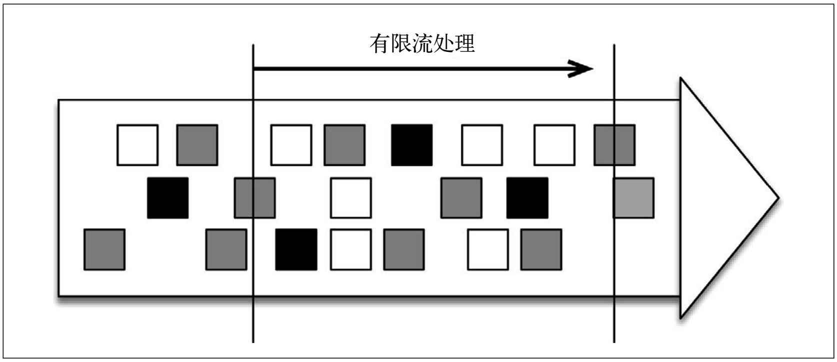



图 6-2：有限流处理：输入数据有头有尾；数据处理在一段时间后停止

显然，有限流处理是无限流处理的一种特殊情况，它只不过在某个时间点停止而已。此外，如果计算结果不在执行过程中连续生成，而仅在末尾处生成一次，那就是*批处理*（分批处理数据）。

批处理是流处理的一种非常特殊的情况。在流处理中，我们为数据定义滑动窗口或滚动窗口，并且在每次窗口滑动或滚动时生成结果。批处理则不同，我们定义一个全局窗口，所有的记录都属于同一个窗口。举例来说，以下代码表示一个简单的 Flink 程序，它负责每小时对某网站的访问者计数，并按照地区分组。

```
val counts = visits
  .keyBy("region")
  .timeWindow(Time.hours(1))
  .sum("visits")
```

如果知道输入数据是有限的，则可以通过以下代码实现批处理。

```
val counts = visits
  .keyBy("region")
  .window(GlobalWindows.create)
  .trigger(EndOfTimeTrigger.create)
  .sum("visits")
```

Flink 的不寻常之处在于，它既可以将数据当作无限流来处理，也可以将它当作有限流来处理。Flink 的 DataSet API 就是专为批处理而生的，如下所示。

```
val counts = visits
  .groupBy("region")
  .sum("visits")
```

如果输入数据是有限的，那么以上代码的运行结果将与前一段代码的相同，但是它对于习惯使用批处理器的程序员来说更友好。

## 6.1　批处理技术

从原则上说，批处理是一种特殊的流处理：当输入数据是有限的，并且只需要得到最终结果时，对所有数据定义一个全局窗口并在窗口里进行计算即可。但是，这样做的效率如何呢？

传统上，有限数据流由专用的批处理器处理；某些时候，它比流处理器更高效。但是，在流处理器中整合高效、大规模的批处理所需的大部分优化方案，是可行的做法。这正是 Flink 所做的工作，而且这样做的效率很高。

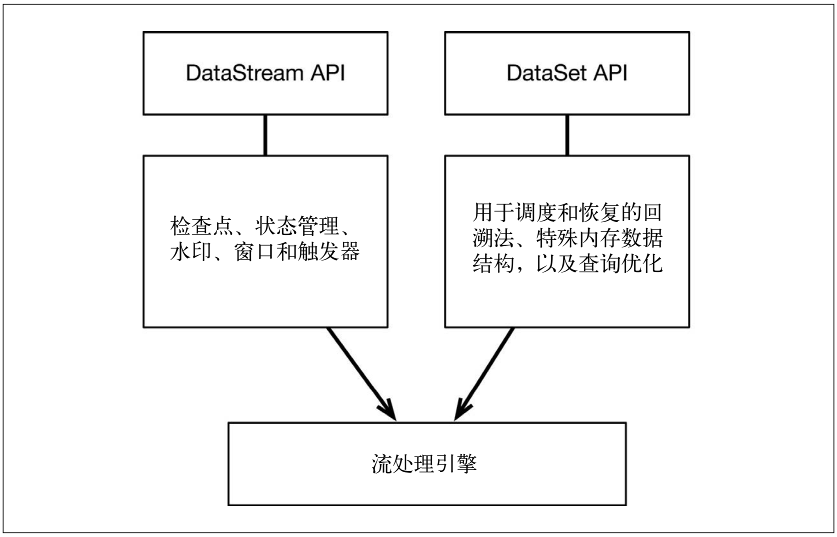



图 6-3：Flink 通过一个底层引擎同时支持流处理和批处理

同样的后端（流处理引擎）被用来处理有限数据和无限数据。在流处理引擎之上，Flink 有以下机制：

- 检查点机制和状态机制：用于实现容错、有状态的处理；
- 水印机制：用于实现事件时钟；
- 窗口和触发器：用于限制计算范围，并定义呈现结果的时间。

在同一个流处理引擎之上，Flink 还存在另一套机制，用于实现高效的批处理。尽管本书并不打算详细讨论这些机制，但是仍要指出以下重点：

- 用于调度和恢复的回溯法：由 Microsoft Dryad 引入，现在几乎用于所有批处理器；
- 用于散列和排序的特殊内存数据结构：可以在需要时，将一部分数据从内存溢出到硬盘上；
- 优化器：尽可能地缩短生成结果的时间。

在写作本书时，以上两套机制分别对应各自的 API（DataStream API 和 DataSet API）；在创建 Flink 作业时，并不能通过将两者混合在一起来同时利用 Flink 的所有功能。然而，这并不是问题。实际上，Flink 社区已经在研究如何统一这两个 API。Apache Beam 社区已经创建出了同时适用于流处理和批处理的 API，它可以生成用于执行的 Flink 程序。

## 6.2 案例研究：Flink作为批处理器

在 2015 年的 Flink Forward 研讨会上，Dongwon Kim[1] 展示了他所做的性能测试。他对 MapReduce、Tez、Spark 和 Flink 在执行纯批处理任务时的性能做了比较。测试的批处理任务是 TeraSort 和分布式散列连接。

第一个任务是 TeraSort，即测量为 1TB 数据排序所用的时间。就上述系统而言，TeraSort 本质上是分布式排序问题，如图 6-4 所示。它由以下几个阶段组成：

(1) 读取阶段：从 HDFS 文件中读取数据分区；
(2) 本地排序阶段：对上述分区进行部分排序；
(3) 混洗阶段：将数据按照 key 重新分布到处理节点上；
(4) 终排序阶段：生成排序输出；
(5) 写入阶段：将排序后的分区写入 HDFS 文件。

注 1：他当时是韩国浦项科技大学的博士后研究员。

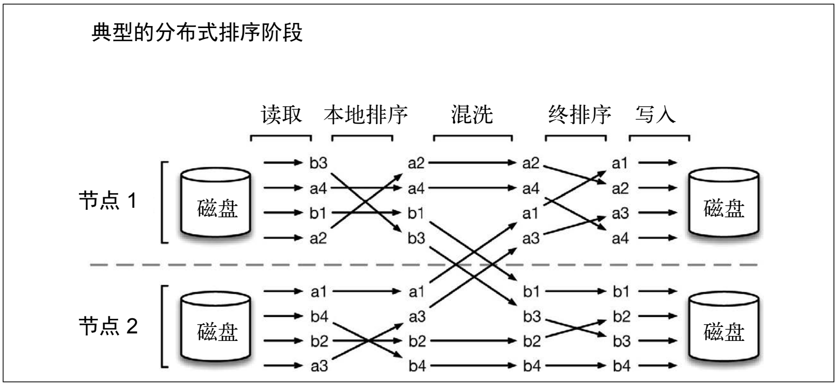



图 6-4：分布式排序的处理阶段

Hadoop 发行版包含对 TeraSort 的实现，同样的实现也可以用于 Tez，因为 Tez 可以执行通过 MapReduce API 编写的程序。Spark 和 Flink 的 TeraSort 实现由 Dongwon Kim 提供[2]。用来测量的集群由 42 台机器组成，每台机器包含 12 个 CPU 内核、24GB 内存，以及 6 块硬盘。

图 6-5 展示了测试结果。结果显示，Flink 的排序时间比其他所有系统都少。MapReduce 用了 2157 秒，Tez 用了 1887 秒，Spark 用了 2171 秒，Flink 则只用了 1480 秒。

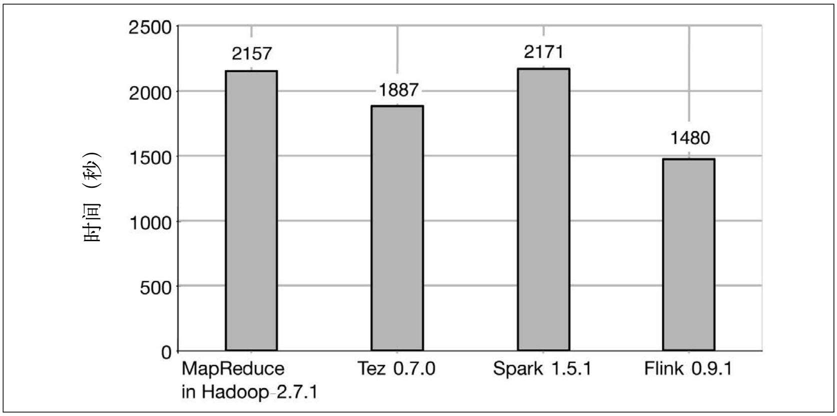



图 6-5：MapReduce、Tez、Spark 和 Flink 分别对应的 TeraSort 结果

注 2：https://github.com/eastcirclek/terasort

第二个任务是一个大数据集（240GB）和一个小数据集（256MB）之间的分布式散列连接。结果显示，Flink 仍然是速度最快的系统，它所用的时间分别是 Tez 和 Spark 的 1/2 和 1/4，如图 6-6 所示。

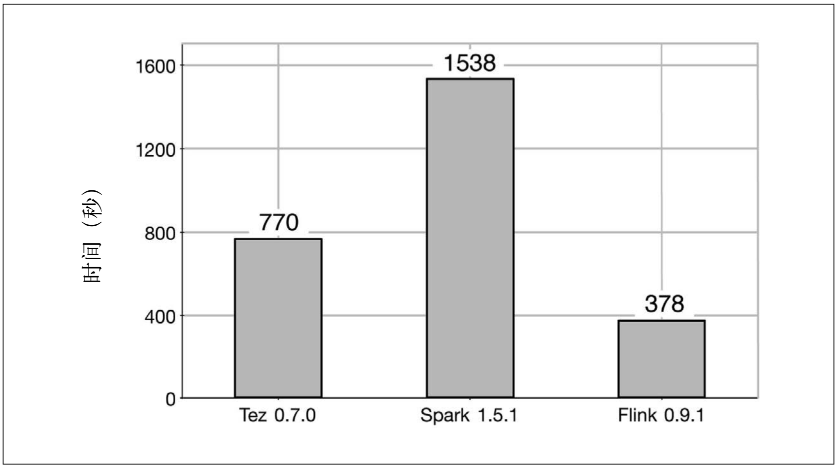



图 6-6：Tez、Spark 和 Flink 分别对应的散列连接结果

产生以上结果的总体原因是，Flink 的执行过程是基于流的，这意味着各个处理阶段有更多的重叠，并且混洗操作是流水线式的，因此磁盘访问操作更少。相反，MapReduce、Tez 和 Spark 是基于批的，这意味着数据在通过网络传输之前必须先被写入磁盘。该测试说明，在使用 Flink 时，系统空闲时间和磁盘访问操作更少。

值得一提的是，性能测试结果中的原始数值可能会因集群设置、配置和软件版本而异。如果现在再测试一遍，那么得到的数值的确可能不一样（上述测试用到的软件版本分别是：Hadoop 2.7.1、Tez 0.7.0、Spark 1.5.1，以及 Flink 0.9.1，这些系统在如今都有了更新的版本）。本节的重点是，即使是批处理器所擅长的任务，流处理器（Flink）在经过适当的优化后也仍然可以表现得和批处理器（MapReduce、Tez 和 Spark）一样好，甚至更好。因此，Flink 可以用同一个数据处理框架来处理无限数据流和有限数据流，并且不会牺牲性能。

# 附录

# 其他资源

## 进一步使用Flink

我们希望你已经跃跃欲试，并且准备好使用 Flink 了。从哪里开始呢？答案是 Flink 的网站：https://flink.apache.org。该网站有“快速入门”指南，通过例子教你如何使用 Flink 摄取和分析维基百科的编辑日志。只需花几分钟，你就可以开始编写你的第一个流处理程序了。

如果你偏爱视觉效果，可以看看 MapR 公司提供的例子：如何用 Flink 摄取纽约市出租车路线的数据流，并用 Kibana 将它可视化，详见 *The Essential Guide to Streaming-first Processing with Apache Flink*（https://www.mapr.com/blog/essential-guide-streaming-first-processing-apache-flink）。

若想进行更深入的学习，可以访问 data Artisans 免费提供的培训资源：http://training.data-artisans.com。其上的所有幻灯片、练习和解决方案都是开源的。

## 关于时间和窗口的更多内容

本书用很大的篇幅讨论了时间和窗口的多个方面，并解释了 Flink 的工作原理以及在使用时可以做的选择。这些主题也是一系列博客文章所讨论的内容。如果你想更深入地了解 Flink 窗口的工作原理，可以参考 http://flink.

apache.org/news/2015/12/04/Introducing-windows.html；若想了解关于会话窗口的更多细节，可以参考 http://data-artisans.com/blog/session-windowing-in-flink/；若想进一步了解 Flink 窗口和水印机制，以及事件时间适用于哪些应用程序，请参考 http://data-artisans.com/blog/how-apache-flink-enables-new-streaming-applications-part-1。

## 关于状态和检查点的更多内容

若想深入了解 Flink 的检查点机制，以及它如何实现容错性流处理，请参考 http://data-artisans.com/blog/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink。

若想了解关于保存点的更多信息，请观看这个短视频：https://mapr.com/blog/savepoints-apache-flink-stream-processing-whiteboard-walkthrough/。在该视频中，data Artisans 的首席技术官 Stephan Ewen 解释了如何用保存点重播流数据。在重新处理数据、修复 bug 和进行更新时，保存点很有用。

更多关于保存点的内容，请参考 http://data-artisans.com/blog/how-apache-flink-enables-new-streaming-applications/。

若想了解基于 Yahoo! Streaming Benchmark 所做的拓展，请访问 https://data-artisans.com/blog/extending-the-yahoo-streaming-benchmark/。

## 用Flink进行批处理

若想了解流处理器如何进行批处理，请访问 http://data-artisans.com/blog/batch-is-a-special-case-of-streaming。

Flink 博客上也有许多信息。若想深入了解 Flink 为了优化批处理而使用的具体机制，请访问以下网址。

- http://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html
- http://flink.apache.org/news/2015/03/13/peeking-into-Apache-Flinks-Engine-Room.html
- http://data-artisans.com/blog/computing-recommendations-at-extreme-scale-with-apache-flink/

## Flink用例和用户故事

经常使用 Flink 的公司会发表一些文章，并在其中描述收获和用法。以下是一些推荐的用户故事。

- https://techblog.king.com/rbea-scalable-real-time-analytics-king/
- https://tech.zalando.de/blog/apache-showdown-flink-vs.-spark/
- http://data-artisans.com/blog/flink-at-bouygues-html/
- http://data-artisans.com/blog/how-we-selected-apache-flink-at-otto-group/

人们每年在 Flink Forward 研讨会上演示的视频和幻灯片，都是了解各个公司如何使用 Flink 的绝佳资源，详见 Flink Forward 的网站。

## 流处理架构

若想深入了解流处理架构，以及 Kafka 和 MapR Streams 所用的消息传输技术，推荐阅读 Ted Dunning 和 Ellen Friedman 合著的书 *Streaming Architecture*。

以下两个短视频解释了流处理架构在支持微服务方面的优势。

- https://www.mapr.com/blog/key-requirements-streaming-platforms-micro-services-advantage-whiteboard-walkthrough-part-1
- https://www.mapr.com/blog/streaming-data-how-move-state-flow-whiteboard-walkthrough-part-2

## 消息传输：Kafka

若想尝试使用 Kafka，可以在 MapR 公司网站上的这篇博客文章里找到示例程序：https://www.mapr.com/blog/getting-started-sample-programs-apache-kafka-09。

此外，推荐阅读由 Neha Narkhede、Gwen Shapira 和 Todd Palino 合著的书《Kafka 权威指南》[1]。

注 1：详见 http://www.ituring.com.cn/book/2067。——编者注

## 消息传输：MapR Streams

MapR Streams 是 MapR 融合数据平台的一个主要部分。要了解关于它的更多内容，请参考下列资源。

- MapR Streams 的功能概览，包括流层面上的管理和跨地域的流复制能力：https://www.mapr.com/products/mapr-streams。
- MapR Streams 的示例程序（用 Kafka API）：https://www.mapr.com/blog/getting-started-sample-programs-mapr-streams。
- 大致比较 Kafka 和 MapR Streams：https://mapr.com/blog/apache-kafka-and-mapr-streams-terms-techniques-and-new-designs。

# Ted Dunning和Ellen Friedman的部分著作

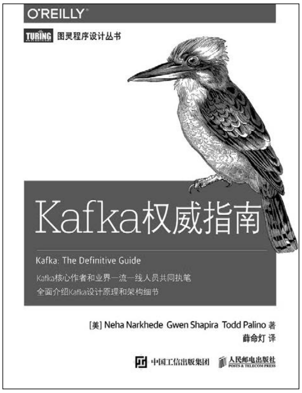
- *Streaming Architecture: New Designs Using Apache Kafka and MapR Streams*
- *Sharing Big Data Safely: Managing Data Security*
- *Real-World Hadoop*
- *Time Series Databases: New Ways to Store and Access Data*
- *Practical Machine Learning: A New Look at Anomaly Detection*
- *Practical Machine Learning: Innovations in Recommendation*

# 关于作者

**埃伦·弗里德曼**（Ellen Friedman）既是解决方案咨询师，也是著名的演讲者和作者。她目前的写作重点是大数据。此外，她还是 Apache Drill 和 Apache Mahout 这两个项目的贡献者。她拥有生物化学博士学位，具有多年的科学研究经验，著作涵盖许多技术主题，包括分子生物学、非传统性遗传方式，以及海洋学。埃伦还是魔法主题卡通书 *A Rabbit Under the Hat* 的合著者。她的 Twitter 用户名是 @Ellen_Friedman。

**科斯塔斯·宙马斯**（Kostas Tzoumas）是 data Artisans 公司的联合创始人兼首席执行官。data Artisans 公司由 Apache Flink 的创始团队创办。科斯塔斯是 Apache Flink 项目管理委员会的成员，拥有丹麦奥尔堡大学的计算机科学博士学位。他是多篇技术论文和博客文章的作者，写作主题是流处理和数据科学。

# 技术改变世界 · 阅读塑造人生

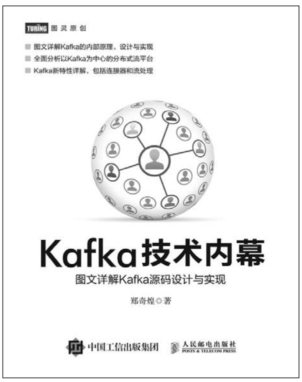



## Kafka 权威指南

- ◆ Kafka核心作者和业界一流一线人员共同执笔
- ◆ 全面介绍Kafka设计原理和架构细节

**书号：** 978-7-115-47327-1

**定价：** 69.00 元

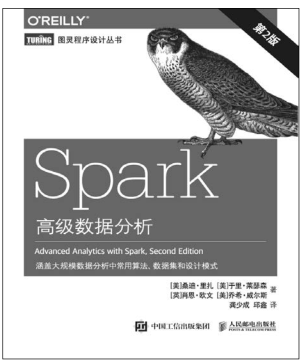



## Kafka 技术内幕

- ◆ 阿里巴巴高级技术专家、Aliware MQ总架构师、Apache RocketMQ联合创始人、Linux OpenMessaging规范发起人冯嘉（Von Gosling），华为云主任工程师时金魁，过往记忆技术博客博主、Qunar数据架构师吴阳平倾力推荐

**书号：** 978-7-115-46938-0

**定价：** 119.00 元



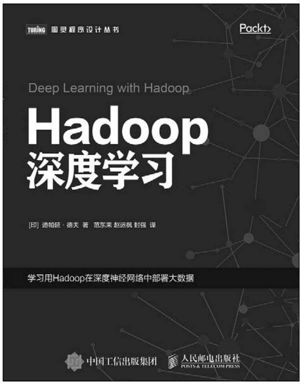

## Spark 高级数据分析（第 2 版）

- ◆ 涵盖大规模数据分析中常用算法、数据集和设计模式

**书号：** 978-7-115-48252-5

**定价：** 69.00 元

# 技术改变世界 · 阅读塑造人生

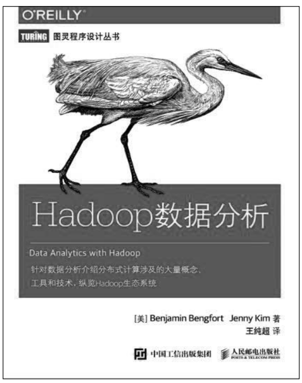



## Hadoop 深度学习

◆ **学习用Hadoop在深度神经网络中部署大数据**

**书号：** 978-7-115-48218-1

**定价：** 39.00 元

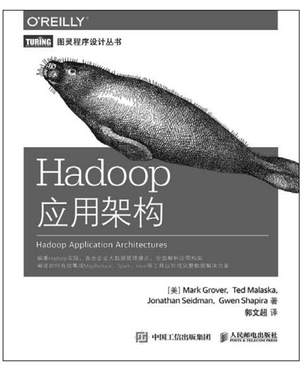



## Hadoop 数据分析

◆ **针对数据分析介绍分布式计算涉及的大量概念、工具和技术，纵览Hadoop生态系统**

**书号：** 978-7-115-47964-8

**定价：** 69.00 元




## Hadoop 应用架构

◆ **偏重Hadoop实践，直击企业大数据管理痛点，全面解析应用架构**

**书号：** 978-7-115-44243-7

**定价：** 69.00 元

O'REILLY®

# Flink基础教程

作为新一代的开源流处理器，Flink是众多大数据处理框架中一颗冉冉升起的新星。它以同一种技术支持流处理和批处理，并能同时满足高吞吐、低延迟和容错的需求。本书由Flink项目核心成员执笔，系统阐释Flink的适用场景、设计理念、功能、用途和性能优势。

- **Flink的适用场景**
- **流处理架构相较于批处理架构的优势**
- **Flink中的时间概念**
- **Flink的检查点机制**
- **Flink的性能优势**


**埃伦·弗里德曼（Ellen Friedman）**，解决方案咨询师，知名大数据相关技术布道师，在流处理架构和大数据处理框架等方面有多部著作。

**科斯塔斯·宙马斯（Kostas Tzoumas）**，Flink项目核心成员，data Artisans公司联合创始人兼首席执行官，在流处理和数据科学领域经验丰富。



封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

| 分类建议 | 计算机／大数据处理 |
|---|---|

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版





ISBN 978-7-115-49006-3

9 787115 490063 >

ISBN 978-7-115-49006-3
定价：39.00元

# 看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks